IN THE UNITED STATES DISTRICT COURT
FOR THE SOUTHERN DISTRICT OF NEW YORK

INTERNATIONAL BUSINESS MACHINES CORPORATION,

Plaintiff,

-vs.-

PLATFORM SOLUTIONS, INC. and T3 TECHNOLOGIES, INC.,

Defendants.

Civil Action No. 06 CV 13565 (LAK)

DECLARATION OF EDWARD J. DEFRANCO IN SUPPORT OF PLAINTIFF IBM'S OPENING CLAIM CONSTRUCTION BRIEF FOR THE *MARKMAN* HEARING

REDACTED VERSION

I, Edward J. DeFranco, under penalty of perjury, declare as follows:

I am a partner at Quinn Emanuel Urquhart Oliver & Hedges, LLP, and I represent Plaintiff International Business Machines Corporation in this case. I submit this Declaration in Support of Plaintiff's Opening Claim Construction Brief for the *Markman* Hearing.

1. Exhibit 1, a true and correct copy of a document entitled "Blueprint Ventures Due Diligence," is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

2. Exhibit 2, a true and correct copy of a document entitled "Soft390™ Product Description and Development Plan," is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

3. Exhibit 3, a true and correct copy of a document entitled "PSI Sales and Technology Review" dated October 2004, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

4. Exhibit 4, a true and correct copy of an e-mail dated March 27, 2003 from George Hoyem to Michael Maulick, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

5. Exhibit 5, a true and correct copy of an e-mail dated November 16, 2005 from Venu Pemmaraju to Bryan Wolf, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

6. Exhibit 6, a true and correct copy of pages 247-250 and 266-268 of the November 28, 2007 deposition of Venu Pemmaraju, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

7. Exhibit 7, a true and correct copy of an e-mail dated September 28, 2006 from Thomas Rosch to george@blueprintventures.com, joe.disabato@gs.com, and rsingh@investcorp.com, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

8. Exhibit 8, a true and correct copy of an e-mail dated March 16, 2003 from Michael Maulick to Ron Hilton, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

9. Exhibit 9, a true and correct copy of an e-mail dated October 30, 2003 from Ron Hilton to Gerry Lane, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

10. Exhibit 10, a true and correct copy of a letter dated February 7, 2003 from Gerald T. Lane to Mr. Ron Hilton, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

11. Exhibit 11, a true and correct copy of an e-mail dated February 7, 2006 from Christian Reilly to Linda Elliot Zider and Michael Maulick with a document attached entitled "PSI Business Update" dated February 7, 2006, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

12. Exhibit 12, a true and correct copy of an e-mail from Gregory Handschuh to Robert Colao and Linda Zider dated July 4, 2004, is being separately filed under seal because it contains information designated as "Confidential" pursuant to the Protective Order entered in this case on June 14, 2007.

13. Attached to this Declaration as Exhibit 13 is a true and correct copy of U.S. Patent No. 5,987,495.

14. Attached to this Declaration as Exhibit 14 is a true and correct copy of U.S. Patent No. 6,775,789.

15. Attached to this Declaration as Exhibit 15 is a true and correct copy of U.S. Patent No. 5,953,520.

16. Attached to this Declaration as Exhibit 16 is a true and correct copy of U.S. Patent No. 6,009,261.

17. Attached to this Declaration as Exhibit 17 is a true and correct copy of U.S. Patent No. 5,687,106.

18. Attached to this Declaration as Exhibit 18 is a true and correct copy of U.S. Patent No. 5,696,709.

19. Attached to this Declaration as Exhibit 19 is a true and correct copy of U.S. Patent No. 5,825,678.

20. Attached to this Declaration as Exhibit 20 is a true and correct copy of U.S. Patent No. 6,654,812.

21. Attached to this Declaration as Exhibit 21 is a true and correct copy of U.S. Patent No. 6,971,002.

22. Attached to this Declaration as Exhibit 22 is a true and correct copy of U.S. Patent No. 5,414,851.

23. Attached to this Declaration as Exhibit 23 is a true and correct copy of page 101 of the book entitled “Built to Last” by Jim Collins.

24. Attached to this Declaration as Exhibit 24 is a true and correct copy of the article entitled “Big Blue Marks Birth of Big Iron” by Daniel Terdiman.

25. Attached to this Declaration as Exhibit 25 is a true and correct copy of pages 2-2 and 2-3 of the 3rd Edition of the “Enterprise System Architecture/390 Principles of Operation” dated December 1994.

26. Attached to this Declaration as Exhibit 26 is a true and correct copy of page 533 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

27. Attached to this Declaration as Exhibit 27 is a true and correct copy of page 292 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

28. Attached to this Declaration as Exhibit 28 is a true and correct copy of page 346 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

29. Attached to this Declaration as Exhibit 29 is a true and correct copy of page 408 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

30. Attached to this Declaration as Exhibit 30 is a true and correct copy of page 539 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

31. Attached to this Declaration as Exhibit 31 is a true and correct copy of page 827 of the IEEE Standard Dictionary of Electrical and Electronics Terms, © 1996.

32. Attached to this Declaration as Exhibit 32 is a true and correct copy of page 566 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

33. Attached to this Declaration as Exhibit 33 is a true and correct copy of a "Response to Office Action" dated July 1, 2002.

34. Attached to this Declaration as Exhibit 34 is a true and correct copy of a "Preliminary Amendment" dated March 3, 2004.

35. Attached to this Declaration as Exhibit 35 is a true and correct copy of Exhibit A to the Joint Claim Construction and Prehearing Statement in this case.

36. Attached to this Declaration as Exhibit 36 is a true and correct copy of U.S. Patent No. 5,732,235.

37. Attached to this Declaration as Exhibit 37 is a true and correct copy of page 1981 of Webster's Third New International Dictionary, © 1993.

38. Attached to this Declaration as Exhibit 38 is a true and correct copy of page 344 of the Microsoft Press Computer Dictionary Second Edition, © 1994.

39. Attached to this Declaration as Exhibit 39 is a true and correct copy of page 107 of the Microsoft Press Computer Dictionary, Second Edition, © 1994.

40. Attached to this Declaration as Exhibit 40 is a true and correct copy of page 663 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

41. Attached to this Declaration as Exhibit 41 is a true and correct copy of page 168 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

42. Attached to this Declaration as Exhibit 42 is a true and correct copy of page 149 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

43. Attached to this Declaration as Exhibit 43 is a true and correct copy of page 523 of The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition, © 2000.

44. Attached to this Declaration as Exhibit 44 is a true and correct copy of page 318 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

45. Attached to this Declaration as Exhibit 45 is a true and correct copy of page 193 of the 10th Edition of the IBM Dictionary of Computing, © 1994.

46. Attached to this Declaration as Exhibit 46 is a true and correct copy of page 298 of The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition, © 2000.

I swear under penalty of perjury that the foregoing is true and correct.

Dated: April 21, 2008

Edward J. DeFranco (ED-6524)

EXHIBIT 13

US005987495A

United States Patent [19]

Ault et al.

[11] Patent Number: 5,987,495

[45] Date of Patent: Nov. 16, 1999

[54] METHOD AND APPARATUS FOR FULLY RESTORING A PROGRAM CONTEXT FOLLOWING AN INTERRUPT

[75] Inventors: Donald F. Ault, Hyde Park; Kenneth E. Plambeck; Casper A. Scalzi, both of Poughkeepsie, all of N.Y.

[73] Assignee: International Business Machines Corporation, Armonk, N.Y.

[21] Appl. No.: 08/966,374

[22] Filed: Nov. 7, 1997

[51] Int. Cl.[6] G06F 9/46

[52] U.S. Cl. 709/108; 710/260

[58] Field of Search 709/108, 300; 710/260, 261, 267; 712/208, 233; 711/200

[56] References Cited

U.S. PATENT DOCUMENTS

4,912,628	3/1990	Briggs	364/200
5,155,853	10/1992	Mitsuhira et al.	395/734
5,161,226	11/1992	Wainer	395/650
5,375,230	12/1994	Fujimori	395/575
5,390,329	2/1995	Gaertner et al.	395/650
5,390,332	2/1995	Golson	395/725
5,428,779	6/1995	Allegrucci et al.	395/650
5,515,538	5/1996	Kleiman	395/733
5,761,492	6/1998	Fernando et al.	395/591
5,790,872	8/1998	Nozue et al.	395/740

FOREIGN PATENT DOCUMENTS

WO9608948 3/1996 European Pat. Off. .

OTHER PUBLICATIONS

IBM Technical Disclosure Bulletin, vol. 32, No. 6B—Nov. 1989 "Deterministic Context Switching of Registers" pp. 70–73.

IBM Technical Disclosure Bulletin, vol. 33, No. 3B, Aug. 1990 "Technique To Improve Context . . . In a CPU", pp. 472–473.

Enterprise Systems Architecture/390 Manual—Principles of Operation SA22-7201-02.

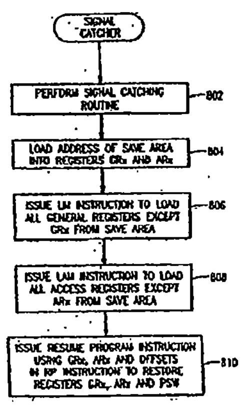
Primary Examiner—Gopal C. Ray

Attorney, Agent, or Firm—William A. Kinnaman, Jr.

[57] ABSTRACT

A method and apparatus for fully restoring the context of a user program, including program status word (PSW) and CPU register contents, following an asynchronous interrupt. Upon the occurrence of an asynchronous interrupt event, control is transferred from the normally executing part of the user program to an interrupt handler of the operating system kernel. The kernel interrupt handler saves the contents of the CPU registers and PSW as they existed at the time of the interrupt in a save area associated with the user program before transferring control to a signal catcher routine of the user program. When it has finished handling the interrupt, the signal catcher routine restores the previous state of program execution as it existed before the interrupt by storing the address of the save area in a selected register (which may be a general register/access register pair), restoring the contents of the registers other than the selected register containing the address of the save area, and then restoring the contents of the PSW and selected register by using a new Resume Program (RP) instruction. The RP instruction contains an operand field specifying through the selected register the base address of the save area together with offset fields specifying the offsets of the saved contents of the PSW and selected register relative to the beginning of the save area. Upon decoding an RP instruction, the CPU executing the instruction adds the displacement to the base address contained in the specified register to form the beginning address of the save area, to which it adds the specified offsets to access the saved PSW and selected register contents. The current PSW and selected register contents are then restored with the saved contents to fully restore the previous program context and return control to the instruction being executed at the point of interrupt. To ensure system integrity, only those fields of the PSW are restored that could have otherwise been restored by a program operating in problem state.

23 Claims, 6 Drawing Sheets



Pc 997081 /LAK

US005987495A

United States Patent [19]

Ault et al.

[11] **Patent Number: 5,987,495**

[45] **Date of Patent: Nov. 16, 1999**

[54] **METHOD AND APPARATUS FOR FULLY RESTORING A PROGRAM CONTEXT FOLLOWING AN INTERRUPT**

[75] Inventors: **Donald F. Ault**, Hyde Park; **Kenneth E. Plambeck**; **Casper A. Scalzi**, both of Poughkeepsie, all of N.Y.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: 08/966,374

[22] Filed: Nov. 7, 1997

[51] Int. Cl.[6] G06F 9/46

[52] U.S. Cl. 709/108; 710/260

[58] Field of Search 709/108, 300; 710/260, 261, 267; 712/208, 233; 711/200

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,912,628	3/1990	Briggs	364/200
5,155,853	10/1992	Mitsuhira et al.	395/734
5,161,226	11/1992	Wainer	395/650
5,375,230	12/1994	Fujimori	395/575
5,390,329	2/1995	Gaertner et al.	395/650
5,390,332	2/1995	Golson	395/725
5,428,779	6/1995	Allegrucci et al.	395/650
5,515,538	5/1996	Kleiman	395/733
5,761,492	6/1998	Fernando et al.	395/591
5,790,872	8/1998	Nozue et al.	395/740

FOREIGN PATENT DOCUMENTS

WO9608948 3/1996 European Pat. Off. .

OTHER PUBLICATIONS

IBM Technical Disclosure Bulletin, vol. 32, No. 6B—Nov. 1989 "Deterministic Context Switching of Registers" pp. 70–73.

IBM Technical Disclosure Bulletin, vol. 33, No. 3B, Aug. 1990 "Technique To Improve Context . . . In a CPU", pp. 472–473.

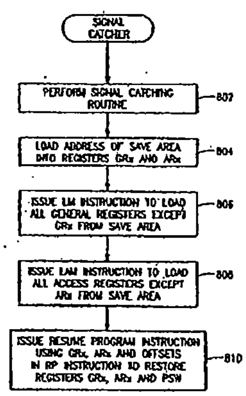
Enterprise Systems Architecture/390 Manual—Principles of Operation SA22-7201-02.

Primary Examiner—Gopal C. Ray
Attorney, Agent, or Firm—William A. Kinnaman, Jr.

[57] **ABSTRACT**

A method and apparatus for fully restoring the context of a user program, including program status word (PSW) and CPU register contents, following an asynchronous interrupt. Upon the occurrence of an asynchronous interrupt event, control is transferred from the normally executing part of the user program to an interrupt handler of the operating system kernel. The kernel interrupt handler saves the contents of the CPU registers and PSW as they existed at the time of the interrupt in a save area associated with the user program before transferring control to a signal catcher routine of the user program. When it has finished handling the interrupt, the signal catcher routine restores the previous state of program execution as it existed before the interrupt by storing the address of the save area in a selected register (which may be a general register/access register pair), restoring the contents of the registers other than the selected register containing the address of the save area, and then restoring the contents of the PSW and selected register by using a new Resume Program (RP) instruction. The RP instruction contains an operand field specifying through the selected register the base address of the save area together with offset fields specifying the offsets of the saved contents of the PSW and selected register relative to the beginning of the save area. Upon decoding an RP instruction, the CPU executing the instruction adds the displacement to the base address contained in the specified register to form the beginning address of the save area, to which it adds the specified offsets to access the saved PSW and selected register contents. The current PSW and selected register contents are then restored with the saved contents to fully restore the previous program context and return control to the instruction being executed at the point of interrupt. To ensure system integrity, only those fields of the PSW are restored that could have otherwise been restored by a program operating in problem state.

23 Claims, 6 Drawing Sheets



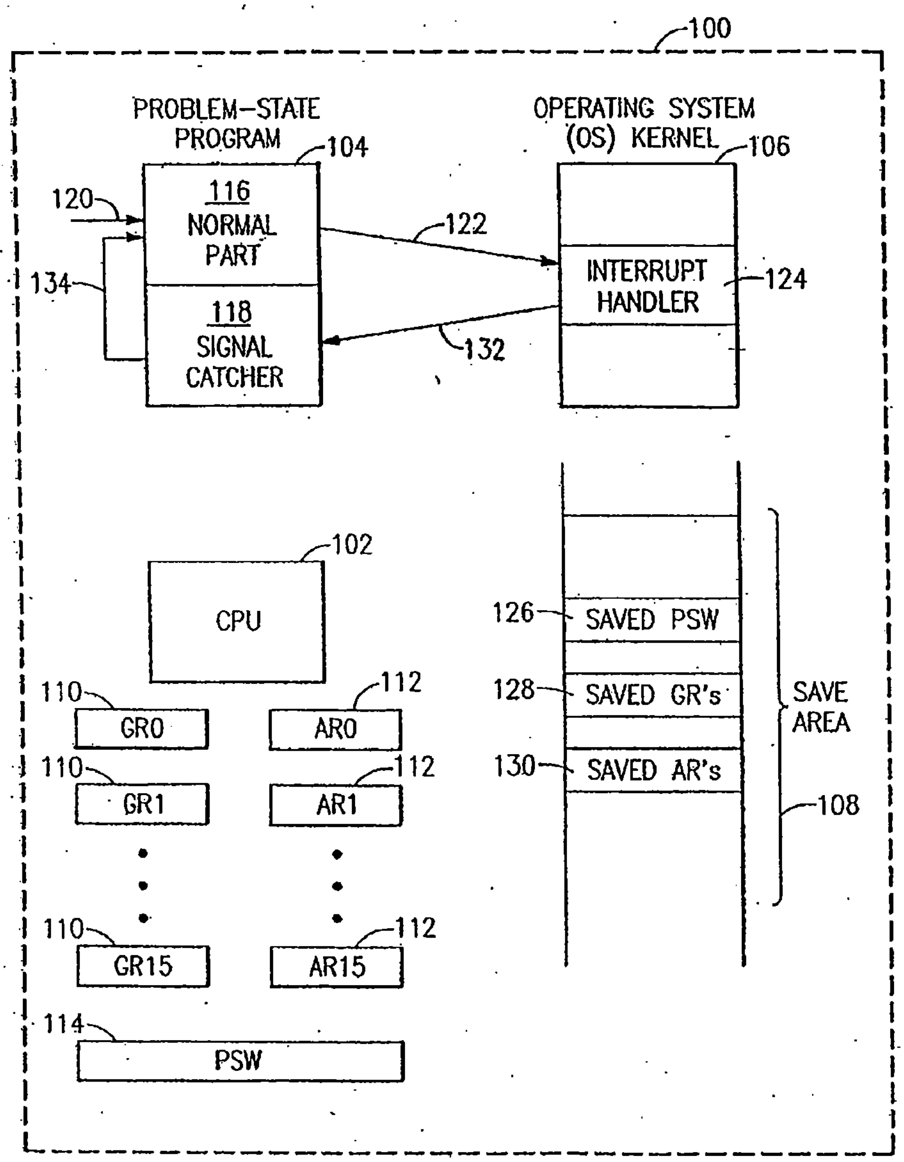
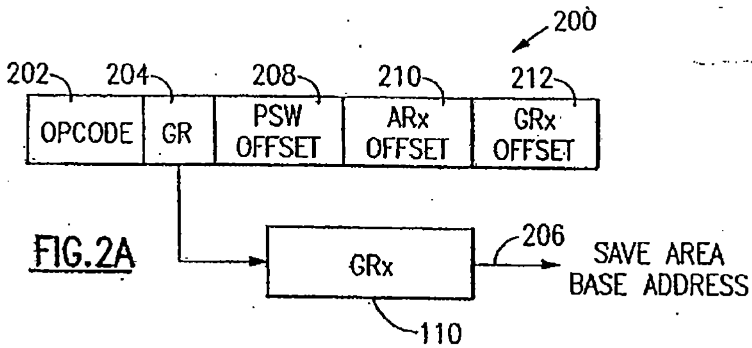
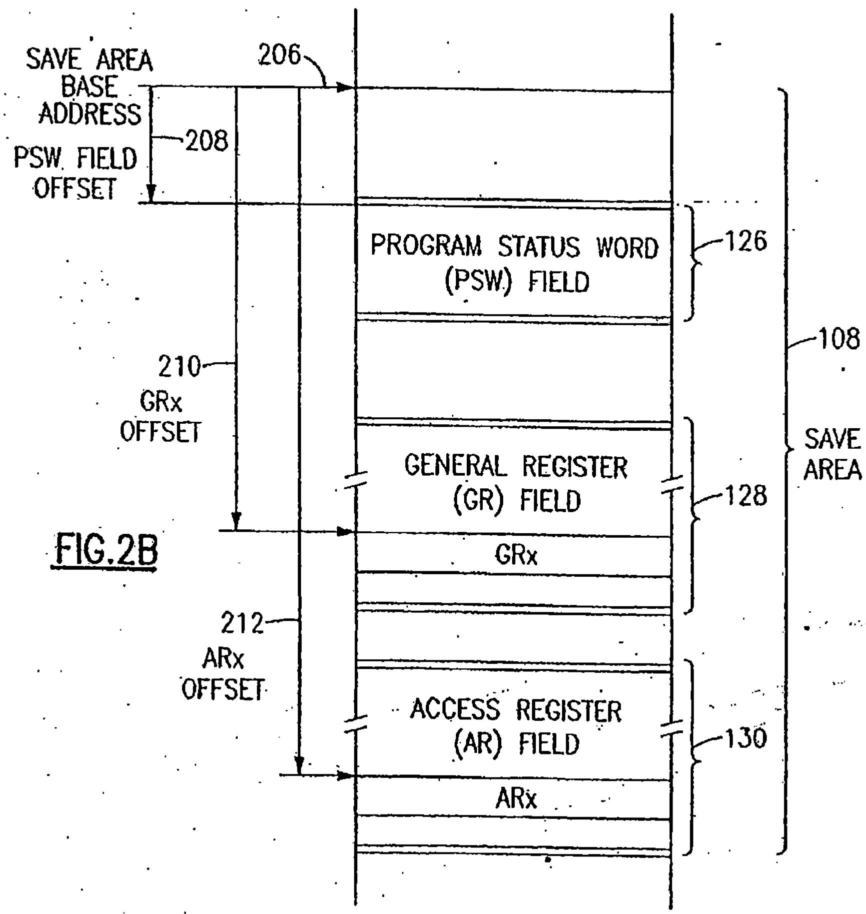

100
PROBLEM-STATE PROGRAM
104
OPERATING SYSTEM (OS) KERNEL
106
120
116 NORMAL PART
122
134
INTERRUPT HANDLER
124
118 SIGNAL CATCHER
132
102
CPU
110
GR0
112
AR0
GR1
AR1
GR15
AR15
114
PSW
126
SAVED PSW
128
SAVED GR's
130
SAVED AR's
SAVE AREA
108

FIG. 1



200
202
204
208
210
212
OPCODE
GR
PSW OFFSET
ARx OFFSET
GRx OFFSET
GRx
206
SAVE AREA BASE ADDRESS
110

FIG.2A



SAVE AREA BASE ADDRESS
206
208
PSW FIELD OFFSET
PROGRAM STATUS WORD (PSW) FIELD
126
108
210
GRx OFFSET
GENERAL REGISTER (GR) FIELD
128
SAVE AREA
GRx
212
ARx OFFSET
ACCESS REGISTER (AR) FIELD
130
ARx

FIG.2B

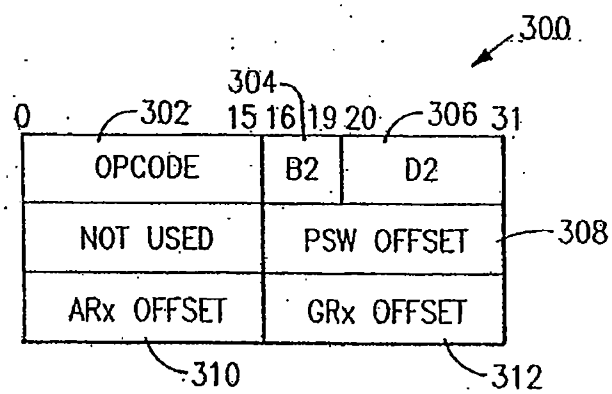

300
304
0
302
15 16
19 20
306
31
OPCODE
B2
D2
NOT USED
PSW OFFSET
308
ARx OFFSET
GRx OFFSET
310
312

FIG.3A

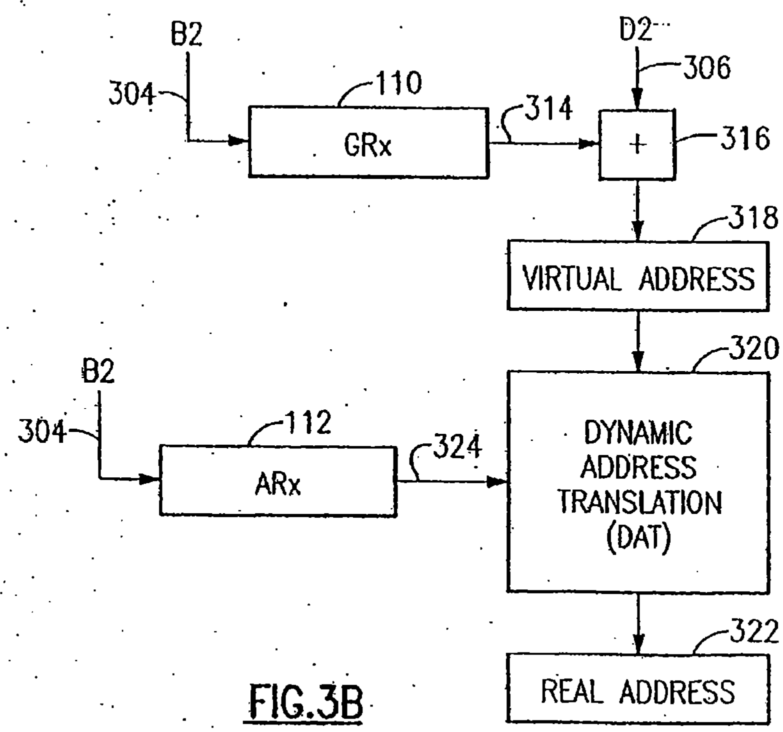

B2
304
110
GRx
314
D2
306
+
316
318
VIRTUAL ADDRESS
320
B2
304
112
ARx
324
DYNAMIC ADDRESS TRANSLATION (DAT)
322
REAL ADDRESS

FIG.3B

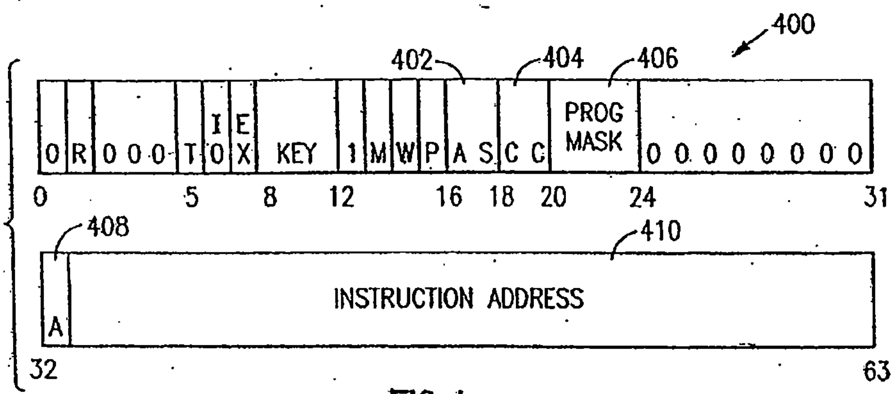
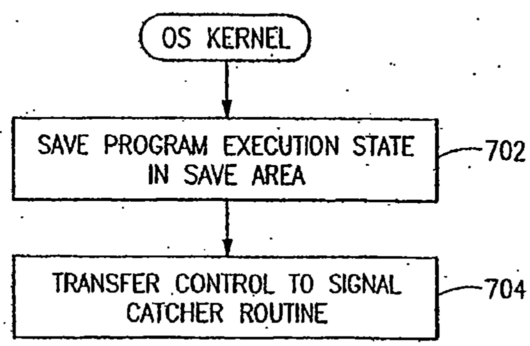


FIG.4

FIG.5

PSW BITS	FIELD NAME
16 AND 17	ADDRESS-SPACE CONTROL (AS)
18 AND 19	CONDITION CODE (CC)
20-23	PROGRAM MASK
32	ADDRESSING MODE (A)
33-63	INSTRUCTION ADDRESS

FIG.7



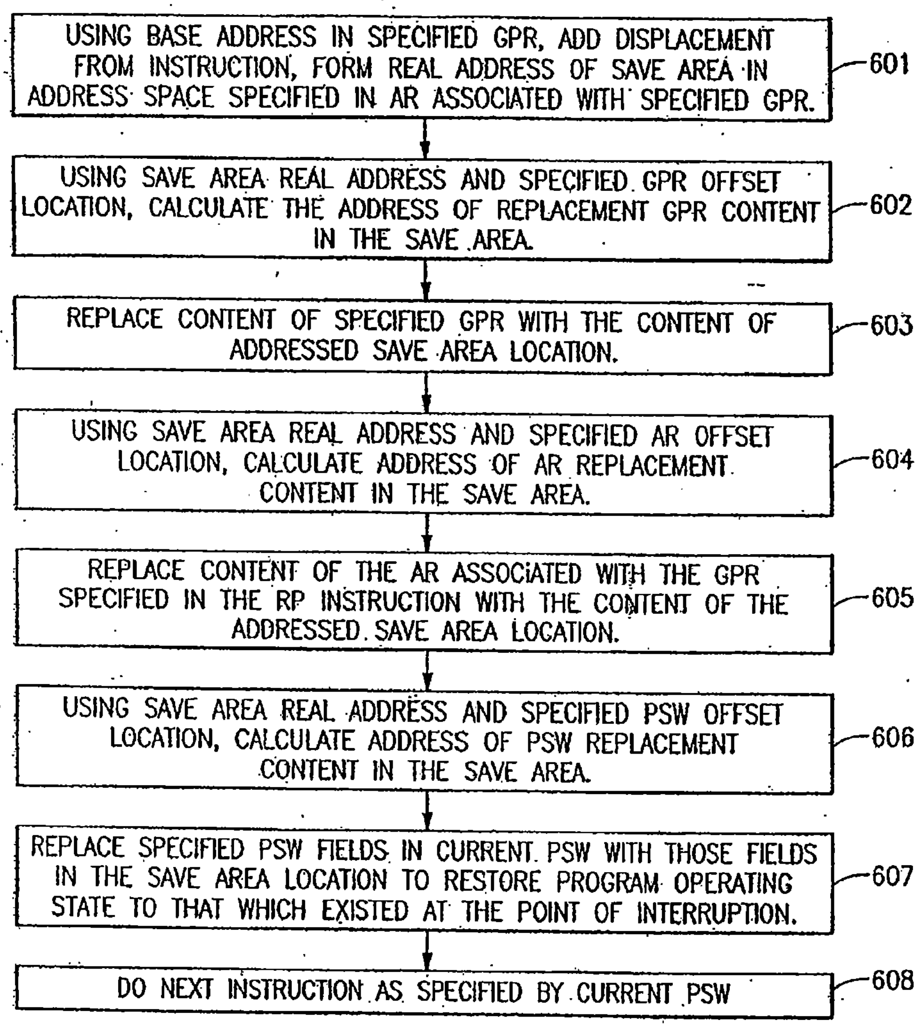

USING BASE ADDRESS IN SPECIFIED GPR, ADD DISPLACEMENT FROM INSTRUCTION, FORM REAL ADDRESS OF SAVE AREA IN ADDRESS SPACE SPECIFIED IN AR ASSOCIATED WITH SPECIFIED GPR.
601
USING SAVE AREA REAL ADDRESS AND SPECIFIED GPR OFFSET LOCATION, CALCULATE THE ADDRESS OF REPLACEMENT GPR CONTENT IN THE SAVE AREA.
602
REPLACE CONTENT OF SPECIFIED GPR WITH THE CONTENT OF ADDRESSED SAVE AREA LOCATION.
603
USING SAVE AREA REAL ADDRESS AND SPECIFIED AR OFFSET LOCATION, CALCULATE ADDRESS OF AR REPLACEMENT CONTENT IN THE SAVE AREA.
604
REPLACE CONTENT OF THE AR ASSOCIATED WITH THE GPR SPECIFIED IN THE RP INSTRUCTION WITH THE CONTENT OF THE ADDRESSED SAVE AREA LOCATION.
605
USING SAVE AREA REAL ADDRESS AND SPECIFIED PSW OFFSET LOCATION, CALCULATE ADDRESS OF PSW REPLACEMENT CONTENT IN THE SAVE AREA.
606
REPLACE SPECIFIED PSW FIELDS IN CURRENT PSW WITH THOSE FIELDS IN THE SAVE AREA LOCATION TO RESTORE PROGRAM OPERATING STATE TO THAT WHICH EXISTED AT THE POINT OF INTERRUPTION.
607
DO NEXT INSTRUCTION AS SPECIFIED BY CURRENT PSW
608

FIG.6

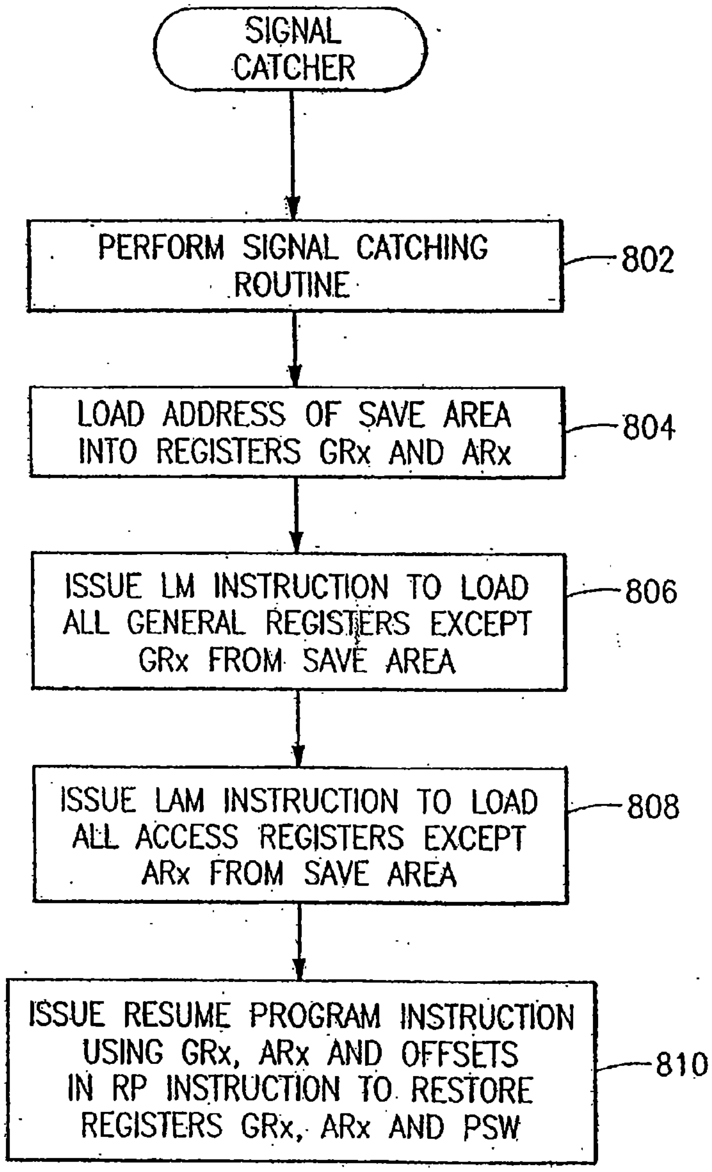

SIGNAL CATCHER
PERFORM SIGNAL CATCHING ROUTINE
802
LOAD ADDRESS OF SAVE AREA INTO REGISTERS GRx AND ARx
804
ISSUE LM INSTRUCTION TO LOAD ALL GENERAL REGISTERS EXCEPT GRx FROM SAVE AREA
806
ISSUE LAM INSTRUCTION TO LOAD ALL ACCESS REGISTERS EXCEPT ARx FROM SAVE AREA
808
ISSUE RESUME PROGRAM INSTRUCTION USING GRx, ARx AND OFFSETS IN RP INSTRUCTION TO RESTORE REGISTERS GRx, ARx AND PSW
810

FIG.8

METHOD AND APPARATUS FOR FULLY RESTORING A PROGRAM CONTEXT FOLLOWING AN INTERRUPT

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to a method and apparatus for fully restoring a program context following an interrupt and, more particularly, to a method and apparatus for restoring the contents of the CPU registers and program status word (PSW) as they existed prior to the asynchronous interrupt of a user program executing in problem state.

2. Description of the Related Art

In a modern computing environment, an operating system is a program (or set of programs) that manages the facilities of a computing system such that the system can be shared among many disparate users being served by multiple independent programs running under the control of that operating system. Hardware resources are under the control of the operating system, which allocates these to the various programs under its control as they request an allocation of them. Thus, real storage space, virtual addressing capability, auxiliary storage space, and ports to the outside world are shared by the programs under the auspices and control of the operating system.

To enforce its management and control over system resources and to provide sharing of facilities with system integrity maintained, the architecture of a system provides mechanisms useful for fencing the operating system from the programs it serves and controls and for fencing those programs from each other. One of these is the operating authority state. Generally, at least two operating states are provided, often called supervisor state and problem state. The supervisor state allows system-wide access authority without fencing, and allows the operating system which uses it to allocate and control which programs have access to which parts of which facilities at which time.

The problem state provides the logical and arithmetic capabilities necessary to solve the problems of a broad range of application programs, and to allow middleware, e.g., database managers or communication access methods, to provide the services to other programs as expected of such middleware. But, in problem state a program is restricted in its accessing capability to that fenced for it by the operating system using the access control mechanisms of the system architecture. These mechanisms are designed to prevent unauthorized access to the operating domain of any other program. Except for performance, and for operating system interfaces expressly provided for intercommunication among programs, the separate programs should not be affected by sharing the system with other programs, and should not be aware of the existence of the other programs. Because of the prevalence of programming error, caused by the complexity of some programming, middleware generally operates in problem state for most of its operating time in order to isolate each such program from the others, in order to minimize the effect of the occasional error, ease detection of the cause of such errors, and improve the recoverability of the system when such errors occur. Further, application programs must be authorized only to those system aspects that affect their own execution. This is particularly true in a world in which computer viruses are seen, and in which, though infrequent, other cases of programming malice are experienced.

Although the present invention may be used in other architectures, it will be discussed in the setting of the IBM® S/390® architecture as documented, for example, in the IBM publication *Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201-02, 1994, and successor versions thereof, incorporated herein by reference.

One of the key mechanisms in an S/390 system is the program status word (PSW), which directs the processor in the execution of a program. It indicates the next instruction to be executed and contains controls constraining the operating state and authority of the program executing under that PSW. Another mechanism is virtual addressing, where the operating system supplies the real backing storage for the virtual storage accessed by problem state programs. Another control mechanism is the set of control words that determine new PSW content on events that must be handled asynchronously by the operating system. For example, when the processor wishes to present a signal that an input/output (I/O) operation has completed and the device or control unit wishes to make a report of the event, this area will indicate the instruction location of the first instruction of the routine that handles the event. The operating system must handle this external event since the I/O devices as a group are shared with different programs allowed access to different ones. The operating system must reflect the completion, in accordance with its own protocols, to programs requiring notification. The PSW content established on the occurrence of an event to be handled by a part of the operating system generally puts the system into the supervisor state, but the interrupted state is saved for later reestablishment when the interrupted program is later to be resumed. Most of the time, the interrupted program was one executing in problem state and restoring state will return the processor to that state. Because the PSW is used to constrain the capability of the program executing on a processor, loading the PSW is restricted to programs executing in supervisor state. One obvious reason is that, depending on the setting of its problem state bit, the PSW authorizes supervisor state or restricts the executing program to problem state with its access and operational restrictions. The restrictions imposed on a problem state program would be of little consequence if the program could simply upgrade its state to supervisor state by overwriting the problem state bit in the PSW.

There are sound technical reasons for allowing a complex program, running in a system with an operating system, to itself contain asynchronous processes associated with asynchronous events for which the program provides special event processing, but to execute in problem program authority state nonetheless, for system integrity reasons. One example occurs in UNIX® programs in which one program may send a message or signal to another program, with the signal arrival occurring asynchronously to the normal processing of the program which is to receive the signal. The kernel program interrupts the normal flow of the program to which it is to deliver the signal and transfers control to a different part of the program designed and coded to handle the asynchronous arrival of the signal. We can call this part of the program its signal catcher routine. The problem posed is that of an efficient return to the normal operating part of the program at its point of interruption after the signal handling part of the program has completed its processing of the signal event. When the operating system kernel handles a logical interruption to an executing program, it has saved the operating state of that program, allowing later resumption of the program, as if the interruption never occurred. In an S/390 system, this involves saving all general purpose resisters (GRs), access registers (ARs), the content of the PSW at the point of interruption, including both the instruction address of the point of interruption and the state

variables controlling the execution of the program. The PSW also records the current setting of the condition code, which reflects the kind of result obtained in the last arithmetic or logical operation, or special circumstances arising in other types of instruction. The program mask, indicating how the processor should behave when certain program exceptions occur during the performance of certain instruction types is also part of the PSW, and actions by the program, which do not require any special authority, can change bits in this field. These are set by the program in concert with its own structure, and each program may have a different program mask and may change it from time to time without communication with the operating system, in order to change the handling of an exception condition. The PSW also specifies the addressing mode, i.e., whether the processor should produce 24-bit addresses or 31-bit addresses when forming effective addresses. This can be changed freely as part of certain branch instructions, so the mode may be either value at any time, and must be restored to that value after an interruption if the program is to operate correctly. The PSW also indicates whether a problem state program is in primary space mode or access register mode at any point in its execution, and this must be properly restored if the program is to execute correctly. Since a problem state instruction can be used to switch between the two addressing modes, the program may be in either mode at any time, unpredictably, and after an interruption, the correct value must be restored.

In the S/390 operational environment, the UNIX kernel itself operates as part of the operating system, and has saved the status of the interrupted program at the point of its interruption. The save area contains the PSW contents as well as the general registers (GRs) and access registers (ARs). Since this save area is provided to allow what is essentially an emulation of an interruption within a single problem state program, it will be preserved should the signal handling part of the program be itself interrupted after it has been entered to handle the signal received. The operating system will use another save area should an interruption occur while the signal handling routine is executing. The save area to be used in returning from the signal handling part of the program to its interrupted part is preserved in storage for that process, in an area accessible by the program itself with its normal storage access authority.

In an S/390 system, which uses the general registers for specifying the addresses of storage operands, it is impossible for a problem state program to transfer control directly to another program using a normal branch instruction and, at the same time, restore all the general registers to some saved earlier value. That is because the save area address is specified by the contents of a general register/access register (GR/AR) register pair, and these values were not the content of the registers at the earlier time of saving the register contents, in the most general case. Also, the branch address must be specified in another general register whose content would generally have been different at the time of saving the registers. Also, it is impossible to properly reflect the control fields of the PSW as they were at the time of the interruption without the use of a Load PSW instruction, which requires the issuing program to be in supervisor state. This is particularly true of the condition code field in the PSW.

The problem has been solved within the operating system since it must perform such actions routinely in dispatching programs. It does this by the use of a PSW in low storage which can be accessed without use of a GR, after disabling the system's interruption capability so that it can not be interrupted in the middle of restoring the execution state of an interrupted program. It would be possible for an operating system service to be defined that would perform the restoration of control back to the interrupted part of the program that the signal catcher is part of, but this would require transition from problem state to supervisor state, and establishment of the PSW to be restored in the low storage area of the computing system, and use of the Load PSW instruction, with the performance negatives of such an instruction path.

What is desired instead is a processor mechanism that provides a direct resumption of an earlier interrupted program without disabling the processor from hardware interruption handling, and without requiring the program to be in an authorized state to cause the resumption from the logical interruption, and without causing a transition to an authorized state to have it done by an authorized system service. It is estimated that such a mechanism would save hundreds, and perhaps even thousands, of executed instructions in doing the program control restoration to the program at the point at which it was logically interrupted for the signal delivery.

SUMMARY OF THE INVENTION

As a preliminary to describing the invention itself, the insights that led to it will be discussed. Problem state programs generally have the least authority among the agencies of the computing system. Only a subset of the architected facilities are available to it. The defined architecture of the system provides the operating system with a set of architected facilities which it may use, and in some cases allocate to problem state programs. Some of those that are withheld from direct use are made available by means of system services provided by the operating system, particularly where physical resources are shared by different programs, e.g., real main storage, external storage space, networking facilities, etc. In like manner, at a lower level of control, the microcode and hardware agencies of the system have defined facilities for their own use which are not accessible, or even seen at the system architecture level. Examples are a section of storage not accessible to programs in either problem or supervisor state, which storage is required to perform the invocable functions of the architecture, registers reserved for internal use, adders and other logical units needed to do addressing and arithmetic. The microcode in particular must perform its assigned programming tasks without polluting the architected facilities of the system. It operates at a third, separate and isolated, level of control in the system, with capabilities beyond the problem state and the supervisor state. The microcode and hardware of the system are designed and the design verified and tested for correctness before the system is manufactured. Special concern is paid that the hardware and microcode can not be compromised as far as system integrity is concerned by actions performed by programs in either supervisor or problem state. Otherwise, the authority structure of the system architecture can not be guaranteed.

The major point here is that the microcode does not change dynamically, it can be considered to be a manufactured component of the system, and is a fully trusted element in the integrity structure of the system. The microcode has full access to programmable storage, and to the Program Status Word (PSW) of the system as part of its necessary capabilities. The microcode can perform complex operations of many steps in order to provide the functions of what is a single instruction at the architecture level above it. Similarly, below the microcode level of control, the hardware elements of the system must have complete access to all system facilities, even those not readily available to the microcode.

The constraint on them is that they may not allow a higher agency to use the facilities usable at the higher level to compromise the integrity of their own operation.

Therefore, in accordance with the present invention, a new instruction, referred to herein as Resume Program (RP), which is invocable in problem state, and is performed at either the microcode or the hardware level, is defined to perform the transition of control from the interrupt-handling routine back to the point of interruption in the main path of the program by restoring state information saved in a save area. The instruction is defined such that only aspects of the PSW that are changeable by a problem state program by other means can be restored using it. Also, since the condition codes of the programming level PSW are not set by the microcode or hardware levels for their own purposes, and the PSW of the programming level is not used by them, they have no problem in establishing a restored state in the PSW.

The instruction addresses the save area by means of a register, and in S/390, possibly an access register (AR), which general register and access register are to be restored by the RP instruction, after using the save area address they specify for the RP instruction itself. The instruction specifies the offsets within the save area of the saved PSW content, the AR to be restored, and the general register to be restored. Only the specified PSW fields (those changeable within problem state) are restored from the saved PSW to the PSW that is given control when the instruction completes its own execution.

More particularly, in one aspect the present invention contemplates a method and apparatus for operating a processor to restore a previously saved program context in an information handling system in which execution of a program by the processor is controlled by a program status word (PSW) defining a program context, in which the program executes in either a first state having relatively restricted authority or a second state having relatively unrestricted authority, and in which the PSW contains a first set of fields that are alterable by a program executing in the first state and a second set of fields that are not alterable by a program executing in the first state. In accordance with this aspect of the invention, a Resume Program (RP) instruction is defined that specifies a storage location containing a saved PSW. Upon decoding an RP instruction, the processor restores from the saved PSW word contained at the specified storage location only those fields of the current PSW that are alterable by a program executing in the first state.

The saved PSW may contain an instruction address that is restored to cause execution to resume at that address. The RP instruction is intended for execution by a program executing in the first state having relatively restricted authority, hence the restrictions on which fields of the PSW are updated. Preferably, the RP instruction contains a field specifying a register, and the storage location is determined using the contents of the specified register. The field may be a first field, and the RP instruction may contain a second field specifying a displacement from a base address, in which case the storage location is determined by adding the displacement contained in the second field to a base address contained in the register specified by the first field.

The save area may also contain the saved contents of the register itself, which are restored to the register from the save area. The RP instruction may specify a beginning address of the save area and an offset from the beginning address, with the storage location being determined by adding the offset specified by the RP instruction to the beginning address of the save area.

Another aspect of the present invention contemplates a method and apparatus for operating a processor to restore a previously saved program context comprising a PSW and a set of register contents. In accordance with this aspect of the invention, the Resume Program (RP) instruction specifies a register selected from the set of registers that points to a save area containing a saved PSW and saved register contents. In an S/390 environment or other environment using access registers in a similar manner, the specified register may be a general register/access register (GR-AR) pair. In response to decoding an RP instruction, the processor accesses the save area using the contents of the specified register and then restores the PSW and the register from the saved PSW and saved register contents contained in the save area. This causes the processor to resume execution at the instruction address contained in the saved PSW with the program context defined by the saved PSW and saved register contents.

As before, the specified register may specify a base address, and the RP instruction may also specify a displacement that is added to the base address to obtain the address of the save area. Likewise, the RP instruction may specify offsets that are added to the beginning address of the save area to obtain the addresses at which the various saved values (PSW, register contents) are stored.

Yet another aspect of the invention contemplates a method whereby a program may use the RP instruction to restore a previously saved PSW and register contents defining a previous program context. In accordance with this aspect of the invention, the program loads the address of the save area containing said previously saved program context into a specified register (which may be a GR/AR pair), restores the contents of the registers of each register type other than the specified register from the save area using a first instruction (e.g., LM for general registers and LAM for access registers in an S/390 environment), and restores the contents of the PSW and the specified register from the save area using the RP instruction to resume execution at the instruction address contained in the saved PSW with the program context defined by the saved PSW and saved register contents.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a computer system in which the present invention may be used.

FIG. 2A depicts the format of a general Resume Program instruction of this invention.

FIG. 2B shows the address arithmetic involved in executing the instruction of FIG. 2A.

FIG. 3A depicts a Resume Program instruction for an S/390 embodiment described herein.

FIG. 3B shows the address arithmetic involved in executing the instruction of FIG. 3A.

FIG. 4 depicts an S/390 Program Status Word (PSW) used in the described embodiment.

FIG. 5 lists the PSW fields that are restored by the execution of the Resume Program (RP) instruction in an S/390 embodiment.

FIG. 6 is a flow chart of the execution of a Resume Program (RP) instruction.

FIG. 7 shows the steps performed by the interrupt handler of the OS kernel.

FIG. 8 shows the steps performed by the signal catcher routine that invokes the Resume Program (RP) Instruction.

DESCRIPTION OF THE PREFERRED EMBODIMENT

FIG. 1 shows a computer system 100 in which the present invention may be used. As shown in the figure, computer

system 100 contains a central processing unit (CPU) 102, at least one user program 104, an operating system (OS) kernel 106, and a save area 108 associated with the user program 104, which has the necessary authorization to access it. Computer system 100 may comprise either a separate physical machine or a separate partition of a logically partitioned machine. Although the invention is not limited to any particular hardware platform, it will be discussed in the exemplary context of an IBM S/390 environment. In such an environment, system 100 may be an IBM S/390 Parallel Enterprise Server, while OS kernel 106 may comprise the IBM OS/390 operating system.

In an S/390 environment, it will be assumed that user program 104 is executing in what is known as the problem state (and is hence referred to as a "problem state program"), while OS kernel 106 operates normally in what is known as the supervisor state. As explained in the S/390 architecture document referred to above, in the supervisor state all instructions are valid, whereas in the problem state only unprivileged instructions and (if certain authority tests are met) semiprivileged instructions are valid.

CPU 102, which constitutes the primary instruction processing unit of system 100, may comprise one or more central processors (CPs) (not separately shown). As is conventional in the art, CPU 102 has an instruction decoder for decoding instructions being executed as well as an execution unit for executing the decoded instructions. These may be implemented by any suitable combination of hardware and microcode in a manner well know in the art. Since the details of their construction and operation form no part of the present invention, they are not separately shown. CPU 102 has an instruction set that (except for the present invention) is generally described in the architecture document referred to above. This instruction set, or architecture, defines how the CPU 102 appears to programming such as user program 104 or OS kernel 106. As noted in the summary portion above, the hardware and microcode implementing the processor architecture, because of their relative immutability, constitute highly "trusted" parts of the system 100, as contrasted with the OS kernel 106 (which is accorded an intermediate level of "trust") or user program 104 (which is accorded the lowest level of "trust").

Associated with CPU 102 are a set of 16 32-bit general registers 110 (GR0–GR15), 16 32-bit access registers 112 (AR0–AR15), and a 64-bit program status word (PSW) 114. General registers 110 are used as base address registers and index registers in address arithmetic and as accumulators in general arithmetic and logical operations. Access registers 112 are used to specify segment table designations used to perform dynamic address translation. PSW 114 stores the address of the next instruction to be executed, along with other pertinent state information, such as a condition code and various settable program modes, as described below. In addition to registers 110 and 112 and PSW 114, CPU 102 has other registers (such as control registers and floating-point registers) that are not relevant to the present invention and are hence not shown.

Problem state program 104 contains a first part 116 that is executed normally and a second part 118 (referred to as a "signal catcher" herein) that is executed in response to an interrupt. More particularly, in response to an asynchronous event at 120 (such as a message or signal from another program), control is transferred at 122 from the problem state program 104 to an interrupt handler 124 of the OS kernel 106 executing in supervisor state. The interruption point at 120 may be arbitrary with respect to the contents of registers 110 and 112 and PSW 114, which cannot be assumed to be any particular value.

In the preferred embodiment, interrupt handler 124 may be a UNIX signal-handling kernel program. Referring also to FIG. 7, upon gaining control, interrupt handler 124 saves the contents of the general registers 110, access registers 112 and portions of PSW 114 (together constituting what will be referred to as the program execution state or program context) as they existed at the point of interrupt in save area 108 (step 702). More particularly, the saved portions of PSW 114 are saved in a saved PSW location 126 of the save area 108, the contents of general registers 110 are saved in a saved GR location 128 of the save area, and the contents of access registers 112 are saved in a saved AR location 130 of the save area. The interrupt handler 124 then transfers control at 132 to the signal catcher routine 118 in the problem state program 104 (step 704).

Upon gaining control, the signal catcher 118 first makes a copy of the passed in save area 108, and then enables recursive signals. That way, each instance of the signal catcher 118 has its own resume save area 108. Thus, signal catcher 118 can be interrupted by a signal, which can be interrupted by another signal, and so on, in a recursive manner as described above.

After this initialization, signal catcher 118 performs its functions (the particulars of which form no part of the present invention) for processing the event that occurred at 120. When signal catcher 118 completes its processing of the interrupt, one of its options is to return at 134 to the point of interruption at 120 to continue normal processing as if an interruption had not occurred.

FIG. 8 shows the steps performed by the signal catcher 118 to return control to the normal part 116 of the program 104 at the point of interruption 120 after the signal catcher has performed its function. Referring to the figure, the signal catcher 118 first loads the address of the save area 108 into a selected general register/access register pair GRx/ARx, where x is an index ranging between 1 and 15 in the embodiment shown (step 802). (In an S/390 environment, x cannot be 0 because GR0 cannot be used for addressing.) Next, the signal catcher 118 restores from the save area 108 all registers 110 and 112 that are not needed by the Resume Program (RP) instruction 300 itself to access the save area. In an S/390 environment, this is done by issuing a Load Multiple (LM) instruction to restore the contents of the set of general registers GRi (where i≠x) from the GR field 128 of the save area 108 (step 804), as well as issuing a Load Access Multiple (LAM) instruction to restore the contents of the set of access registers GRi (where i≠x) from the AR field 130 of the save area 108 (step 806). Although step 804 is shown as preceding step 806 in FIG. 8, the particular order in which the steps are performed is immaterial.

At this point the signal catcher 118 invokes the Resume Program (RP) instruction 300 of the present invention to do full context restoration to the point of the interruption (step 808). The RP instruction 300 uses the content of the general register GRx (and access register ARx, if needed) specified in the instruction to access the save area 108, in the manner described in detail below.

FIG. 2A shows the logical format 200 of the Resume Program (RP) instruction of the present invention, which is not specific to any particular platform. In the logical format 200, an operation code (OPCODE) 202 identifies the instruction as an RP instruction; a register specification (GR) 204 specifies a general register 110 (GRx) that contains the base address 206 of the save area 108; a PSW offset 208 specifies the offset of the saved PSW contents 126 from the beginning of the save area 108; an ARx offset 210 specifies

the offset of the saved ARx contents from the beginning of the save area 108; and a GRx offset 212 specifies the offset of the saved GRx contents from the beginning of the save area 108.

FIG. 2B shows graphically the address arithmetic involved in using the operands 204–212. As shown in the figure, to generate the beginning address of the PSW field 126, the PSW offset 208 is added to the base address 206 contained in the general register 110 (GRx) pointed to by the GR field 204. Similarly, to generate the address of the saved GRx contents in field 128, the GRx offset 210 is added to the base address 206 contained in the general register GRx. Finally, to generate the address of the saved ARx contents in field 130, the ARx offset 212 is added to the base address 206 contained in the general register GRx. Although not shown in FIGS. 2A and 2B, the base address 206 contained in general register GRx may be a virtual address that is converted into a real address by dynamic address translation (DAT). In such a case, the corresponding access register ARx may be used to specify a particular address space for which the conversion is performed or to otherwise control the address translation.

FIG. 3A illustrates a possible instruction format 300 for an S/390 environment, while FIG. 3B shows the additional address arithmetic implied by the format. Format 300 comprises a 32-bit instruction proper (bits 0–31), followed immediately by a 64-bit parameter list that for practical purposes may be regarded as part of the instruction. The instruction proper contains a 16-bit opcode field 302, followed by a 4-bit base register field 304 (B2) and a 12-bit displacement field 306 (D2). The 64-bit parameter list contains a 16-bit unused field (filled with zeros), followed by a 16-bit PSW offset 308, a 16-bit ARx offset 310 and a 16-bit GRx offset 312.

Fields 302 and 308–312 are similar to the corresponding fields 202 and 208–212 in logical format 200 and will not be redescribed. Field 304 (B2), like field 204 in logical format 200, specifies a particular general register 110 (GRx) used to point to the save area 108. However, the address contained in the specified register GRx, rather than pointing directly to the beginning address of save area 108, is a base address 314 that is combined at 316 with a displacement D2 specified in field 306 to obtain a beginning address 318 for the save area. As with the address 206, beginning address 318 may be a virtual address that is converted by dynamic address translation (DAT) 320 to a real address 324. As suggested above for the logical format 200, the dynamic address translation 320 may depend on an address space specification determined by an input 324 from the corresponding access register 112 (ARx). The particulars of the dynamic address translation 320 are described in the S/390 architecture document identified above. Except for the fact that the dynamic address translation 320 is determined in part by the contents 324 of the access register ARx, its particulars form no part of the present invention and are hence not discussed in this specification.

As described below, execution of the RP instruction 300 causes certain fields in the current PSW 114 and the contents of the access register 112 (ARx) and general register 110 (GRx) specified by the index in field 304 (B2) to be replaced with fields in the save area 108 (as specified by the second operand address B2, D2) having the specified offsets 308–312.

FIG. 4 shows the format of a conventional S/390 program status word (PSW) 400, as described, for example, in the architecture document referred to above. PSW 400 contains several fields of interest to the present invention, since they are saved in the PSW portion 126 of save area 108 following an interrupt. These fields include an address space control (AS) 402 (bits 16–17); a condition code (CC) 404 (bits 18–19); a program mask 406 (bits 20–23); an addressing mode (A) 408 (bit 32); and an instruction address 410 (bits 33–63). The address space control (AS) 402 specifies, in conjunction with fields in the control registers and control blocks, the instruction address space and the address space containing storage operands. The condition code (CC) 404 is set as a result of certain arithmetic operations and comparisons and can be used to do conditional branching so as to direct program flow based on past results. The program mask 406 specifies, for certain arithmetic results, whether or not those results should cause an interruption, either for terminating program execution or for modifying the results. The addressing mode (A) 408 specifies either a 24-bit or a 31-bit addressing mode. The instruction address 410 is the address of the next instruction to be executed.

PSW 400 contains other fields, which are not restored by the Resume Program instruction of the present invention, since they are not alterable by a program executing in problem state. These include bits 0–15, of which bit 15 is the problem state (P) bit defining whether the CPU 102 is in the problem state (P=1) or in the supervisor state (P=0).

FIG. 5 lists the fields of PSW 400 that are restored by Resume Program from the PSW in the save area used by Resume Program in an S/390 embodiment. As indicated above, the restored fields include the address space control 402, the condition code 404, the program mask 406, the addressing mode 408, and the instruction address 410.

FIG. 6 is a flowchart of the operation of the Resume Program (RP) instruction 300. These actions are performed at the microcode and hardware level of the system 100, within CPU 102; the particulars of their implementation at these levels form no part of the present invention and are hence not shown. Changes to the architected level can not be seen by the program 104 until the RP instruction 300 has completed and control is returned to the program 104 using the PSW 114 as modified by the RP instruction.

Upon decoding an RP instruction, CPU 102 obtains the real address 322 (FIG. 3B) of the save area 108 by forming the effective virtual address 318 and performing normal address translation 320 (step 601).

Next, the address in the save area 108 of the saved value of the general register GRx (specified by the B2 field 304 of the RP instruction) is calculated using the base address 322 of the save area and the GRx offset 312 specified in the instruction parameter list (step 602). This address is then used to replace the content of the general register GRx with the saved content at the addressed save area location (step 603).

The procedure of steps 602–603 is then repeated for the access register ARx associated with the general register GRx. Using the save area real address 322 and the ARx offset 310 specified in the parameter list, the address in the save area 108 of the saved content of the access register ARx is calculated (step 604). The content of the access register ARx is then replaced with the content from the addressed save area location (step 605).

The procedure is then repeated once again to restore the PSW 114. The real address of the PSW 126 in the save area 108 is calculated by using the save area real address 322 and the PSW offset 308 specified in the RP parameter list (step 606). Then, the fields specified for change in the RP instruction definition (FIGS. 4–5) are replaced by the correspond-

ing fields in the stored PSW 126 in the save area 108 (step 607). These fields include the instruction address 410 of the next instruction to be executed in the interrupted part 116 of the program 104; therefore, the RP instruction 300 in effect causes a branch.

Finally, following execution of the RP instruction 300, the next instruction in the interrupted program 104 is executed as specified by the restored PSW 114 (step 608).

As already noted, the RP instruction described above, like other aspects of the CPU architecture, may be implemented by hardware, by microcode, or by any suitable combination of the two, while the signal catcher 118 utilizing the RP instruction is preferable implemented as software. (Both microcode and software constitute programming, the principal difference being that microcode implements an architectural interface while software interacts with it.) While a particular embodiment has been shown and described, those skilled in the art will appreciate that various modifications may be made without departing from the principles of the invention.

What is claimed is:

1. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word defining a program context, said program executing in either a problem state having relatively restricted authority or a supervisor state having relatively unrestricted authority, said program status word containing a first set of fields that are alterable by a program executing in said problem state and a second set of fields that are not alterable by a program executing in said problem state, a method of operating said processor to restore a previously saved program context, comprising the steps of:

decoding an instruction from a program executing in said problem state specifying a storage location containing a saved program status word; and

in response to decoding said program instruction; restoring from the saved program status word contained at said specified storage location only those fields of the current program status word that are alterable by a program executing in said problem state.

2. The method of claim 1 in which said program instruction contains a field specifying a register, said restoring step comprising the step of:

determining said storage location using the contents of the register specified by said field.

3. The method of claim 2 in which said field is a first field, said program instruction containing a second field specifying a displacement from a base address, said restoring step comprising the step of:

determining said storage location by using the contents of the register specified by said first field as a base address and adding to said base address the displacement contained in said second field.

4. The method of claim 2 in which said register specifies a save area containing said storage location and saved contents of said register, said restoring step comprising the further step of:

restoring said saved register contents to said register from said save area.

5. The method of claim 1 in which said program instruction specifies a beginning address of a save area and an offset from said beginning address, said restoring step comprising the step of:

determining said storage location by adding the offset specified by said program instruction to the beginning address of the save area specified by said program instruction.

6. The method of claim 1 in which said saved program status word contains an instruction address that is restored by said restoring step to cause execution to resume at said instruction address.

7. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word and by a set of registers defining a program context, said program status word containing an instruction address, a method of operating said processor to restore a previously saved program context, comprising the steps of:

decoding a program instruction specifying a register selected from said set of registers, said register pointing to a save area containing a saved program status word and saved register contents, and

in response to decoding said program instruction:

accessing said save area using the contents of the register specified by said program instruction; and

restoring said program status word and said register from the saved program status word and saved register contents contained in said save area to resume execution at the instruction address contained in said saved program status word with the program context defined by said saved program status word and saved register contents.

8. The method of claim 7 in which said program instruction specifies a general register.

9. The method of claim 7 in which said program instruction specifies a general register and an access register, said accessing step using the contents of said general register and said access register to access said save area.

10. The method of claim 7 in which the specified register specifies a base address, said program instruction also specifying a displacement from said base address, said accessing step comprising the step of:

adding the specified displacement to the base address contained in said specified register.

11. The method of claim 7 in which said register specifies a beginning address of said save area, said program instruction also specifying an offset from said beginning address, said accessing step comprising the step of:

adding the offset specified by said program instruction to the beginning address of the save area specified by said register.

12. The method of claim 7 in which said program executes in either a problem state having relatively restricted authority or a supervisor state having relatively unrestricted authority, said restoring step being performed for a program executing in said problem state and restoring only those fields of the current program status word that are alterable by a program executing in said problem state.

13. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word and by a set of registers defining a program context, said program status word containing an instruction address, a method of restoring a previously saved program status word and saved register contents defining a previous program context, comprising the steps of:

loading the address of a save area containing said previously saved program context into a specified one of said registers;

restoring the contents of said registers other than said specified register from said save area using a first instruction; and

restoring the contents of said program status word and said specified register from said save area using a

second instruction to resume execution at the instruction address contained in said saved program status word with the program context defined by said saved program status word and saved register contents.

14. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word defining a program context, said program executing in either a problem state having relatively restricted authority or a supervisor state having relatively unrestricted authority, said program status word containing a first set of fields that are alterable by a program executing in said problem state and a second set of fields that are not alterable by a program executing in said problem state, apparatus for operating said processor to restore a previously saved program context, comprising:

means for decoding an instruction from a program executing in said problem state specifying a storage location containing a saved program status word; and

means responsive to said decoding means for restoring from the saved program status word contained at said specified storage location only those fields of the current program status word that are alterable by a program executing in said problem state.

15. The apparatus of claim 14 in which said program instruction contains a field specifying a register, said restoring means comprising:

means for determining said storage location using the contents of the register specified by said field.

16. The apparatus of claim 15 in which said field is a first field, said program instruction containing a second field specifying a displacement from a base address, said restoring means comprising:

means for determining said storage location by using the contents of the register specified by said first field as a base address and adding to said base address the displacement contained in said second field.

17. The apparatus of claim 15 in which said register specifies a save area containing said storage location and saved contents of said register, said restoring means further comprising:

means for restoring said saved register contents to said register from said save area.

18. The apparatus of claim 14 in which said program instruction specifies a beginning address of a save area and an offset from said beginning address, said restoring means comprising:

means for determining said storage location by adding the offset specified by said program instruction to the beginning address of the save area specified by said program instruction.

19. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word and by a set of registers defining a program context, said program status word containing an instruction address, apparatus for operating said processor to restore a previously saved program context, comprising:

means for decoding a program instruction specifying a register selected from said set of registers, said register pointing to a save area containing a saved program status word and saved register contents, and

means response to said decoding means for executing said instruction, said executing means comprising:

means for accessing said save area using the contents of the register specified by said program instruction; and

means for restoring said program status word and said register from the saved program status word and saved register contents contained in said save area to resume execution at the instruction address contained in said saved program status word with the program context defined by said saved program status word and saved register contents.

20. The apparatus of claim 19 in which the specified register specifies a base address, said program instruction also specifying a displacement from said base address, said accessing means comprising:

means for adding the specified displacement to the base address contained in said specified register.

21. The apparatus of claim 19 in which said register specifies a beginning address of said save area, said program instruction also specifying an offset from said beginning address, said accessing means comprising:

means for adding the offset specified by said program instruction to the beginning address of the save area specified by said register.

22. The apparatus of claim 19 in which said program executes in either a problem state having relatively restricted authority or a supervisor state having relatively unrestricted authority, said restoring means being operative for a program executing in said problem state and restoring only those fields of the current program status word that are alterable by a program executing in said problem state.

23. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word and by a set of registers defining a program context, said program status word containing an instruction address, apparatus for restoring a previously saved program status word and saved register contents defining a previous program context, comprising:

means for loading the address of a save area containing said previously saved program context into a specified one of said registers;

means for restoring the contents of said registers other than said specified register from said save area using a first instruction; and

means for restoring the contents of said program status word and said specified register from said save area using a second instruction to resume execution at the instruction address contained in said saved program status word with the program context defined by said saved program status word and saved register contents.

* * * * *

EXHIBIT 14

US006775789B2

(12) United States Patent
Elko et al.

(10) Patent No.: US 6,775,789 B2
(45) Date of Patent: Aug. 10, 2004

(54) METHOD, SYSTEM AND PROGRAM PRODUCTS FOR GENERATING SEQUENCE VALUES THAT ARE UNIQUE ACROSS OPERATING SYSTEM IMAGES

(75) Inventors: David Arlen Elko, Austin, TX (US); Jeffrey M. Nick, West Park, NY (US); Ronald M. Smith, Sr., Wappingers Falls, NY (US); Charles F. Webb, Poughkeepsie, NY (US)

(73) Assignee: International Business Machines Corporation, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: 09/337,158

(22) Filed: Jun. 21, 1999

(65) **Prior Publication Data**

US 2003/0101365 A1 May 29, 2003

(51) Int. Cl.[7] G06F 1/04; G06F 1/14
(52) U.S. Cl. 713/500; 713/600
(58) Field of Search 709/400, 248; 713/400, 500, 600, 1, 2; 702/125, 187

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,257,379 A * 10/1993 Cwiakala et al. 710/7
5,317,739 A 5/1994 Elko et al.
5,416,921 A * 5/1995 Frey et al. 714/11
5,561,809 A 10/1996 Elko et al.
5,706,432 A 1/1998 Elko et al.
5,933,625 A * 8/1999 Sugiyama 713/503
6,363,389 B1 * 3/2002 Lyle et al. 707/1

OTHER PUBLICATIONS

P. J. Wanish, Jan. 1981, IBM Technical Disclosure Bulletin, vol. 23, No. 8, pp 3819–3820.*
"Enterprise Systems Architecture/390 Principles of Operation", IBM Publication No. SA22-7201-04, Fifth Edition (Jun. 1997), pp. 4-25-4-32; 4-38; 6-11-6-12; 7-81-7/82; 10-69; 11-10; 12-5.

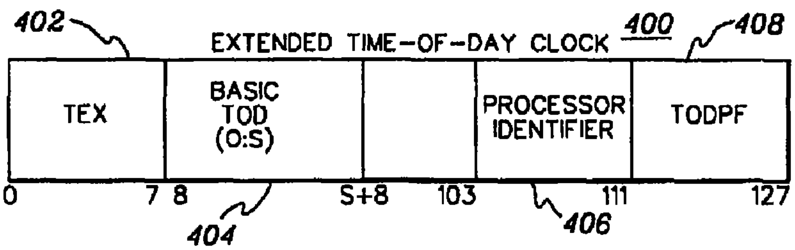
* cited by examiner

Primary Examiner—Ilwoo Park
(74) *Attorney, Agent, or Firm*—Floyd A. Gonzales, Esq.; Marc A. Ehrlich, Esq.; Heslin Rothenberg Farley & Mesiti P.C.

(57) **ABSTRACT**

Timing facilities are used to provide sequence values that are unique across operating system images. A sequence value includes various components, including timing information and selected information. The selected information is used to provide a sequence value that is unique across a plurality of operating system images. Additionally, the sequence value can include, for instance, a processor identifier component and a placeholder component. The placeholder component ensures that the sequence value is an increasing value, even when the physical clock used to provide the timing information wraps back to zero.

41 Claims, 8 Drawing Sheets



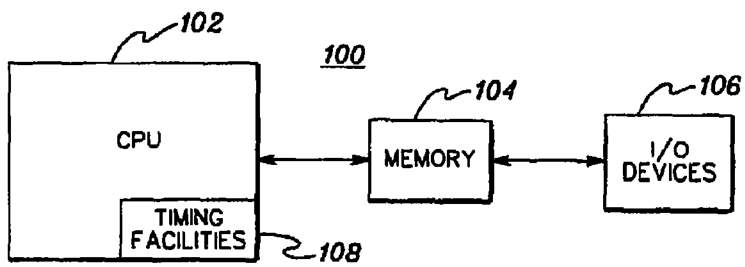

102
100
104
106
CPU
MEMORY
I/O DEVICES
TIMING FACILITIES
108

fig. 1

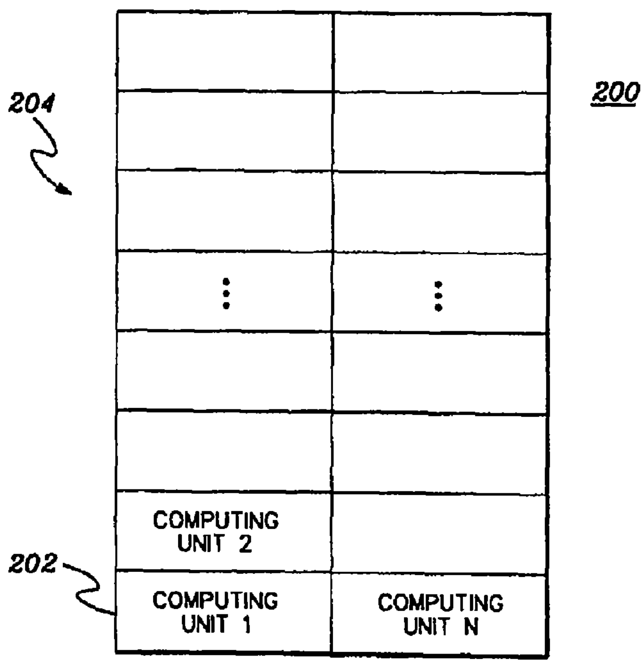

204
200
COMPUTING UNIT 2
202
COMPUTING UNIT 1
COMPUTING UNIT N

fig. 2

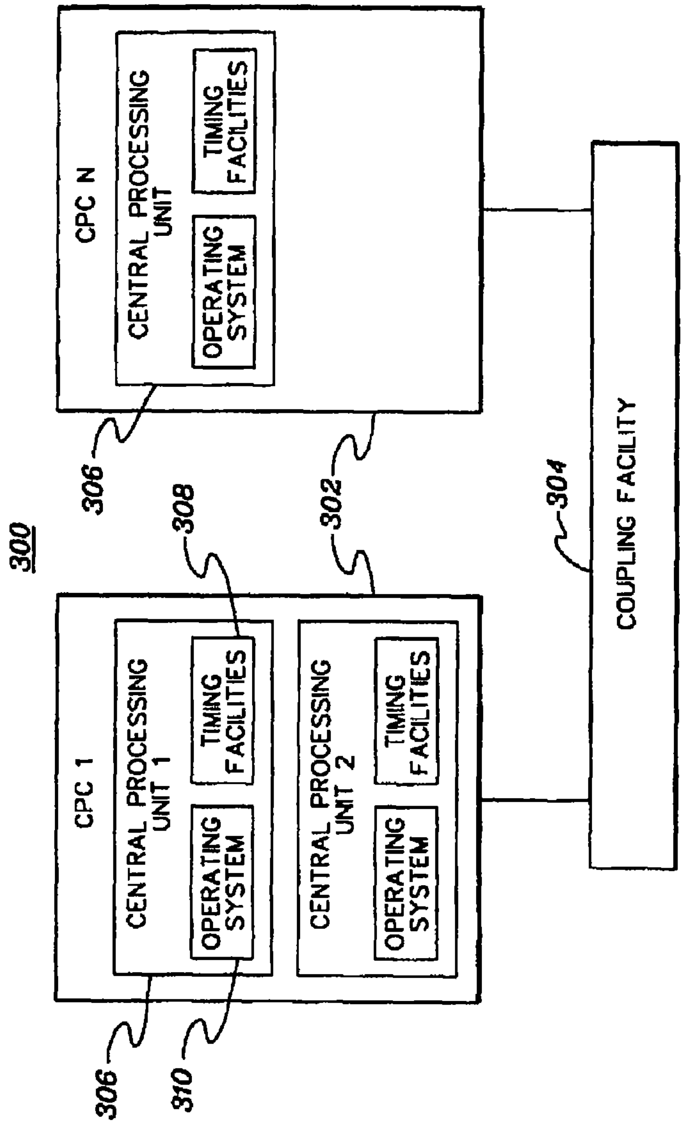

300
CPC N
CENTRAL PROCESSING UNIT
OPERATING SYSTEM
TIMING FACILITIES
306
308
302
304
COUPLING FACILITY
CPC 1
CENTRAL PROCESSING UNIT 1
OPERATING SYSTEM
TIMING FACILITIES
CENTRAL PROCESSING UNIT 2
OPERATING SYSTEM
TIMING FACILITIES
306
310

fig. 3

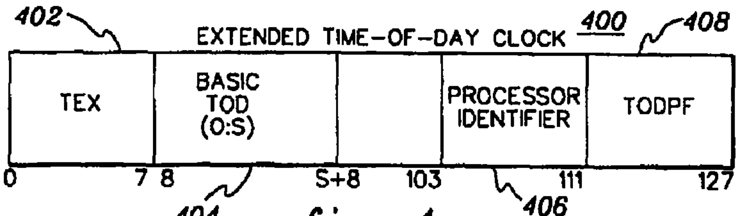
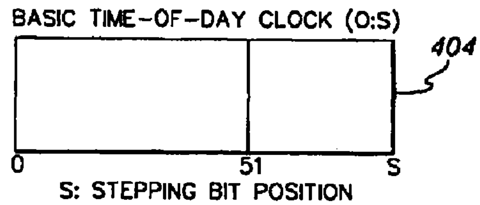


fig. 4

fig. 5



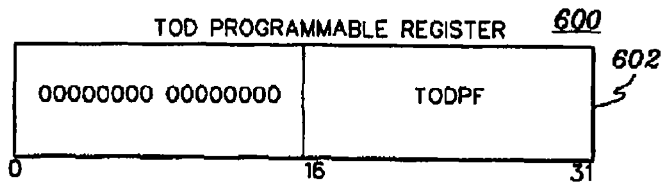
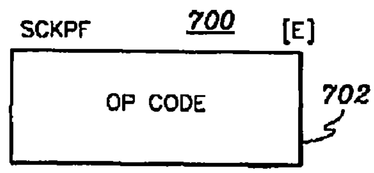


fig. 6

fig. 7



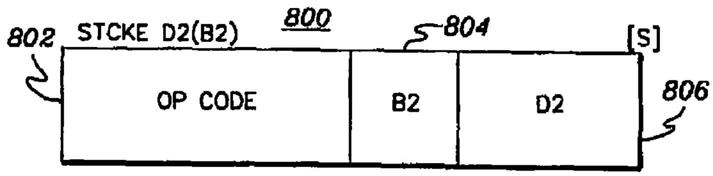


fig. 8

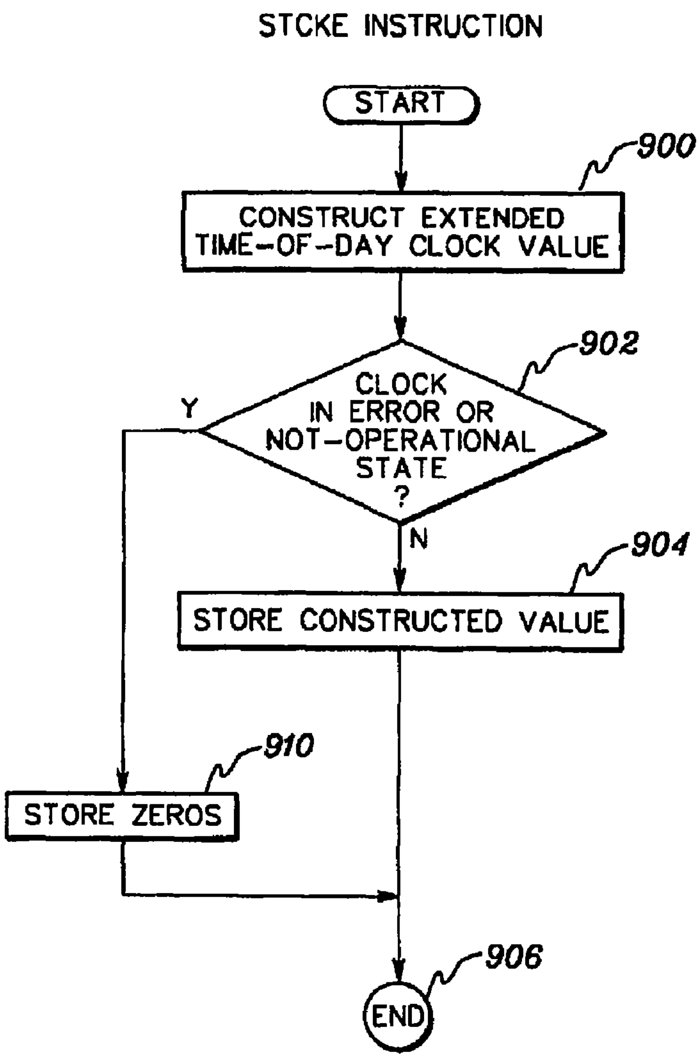

STCKE INSTRUCTION
START
900
CONSTRUCT EXTENDED TIME-OF-DAY CLOCK VALUE
902
CLOCK IN ERROR OR NOT-OPERATIONAL STATE ?
Y
N
904
STORE CONSTRUCTED VALUE
910
STORE ZEROS
906
END

fig. 9

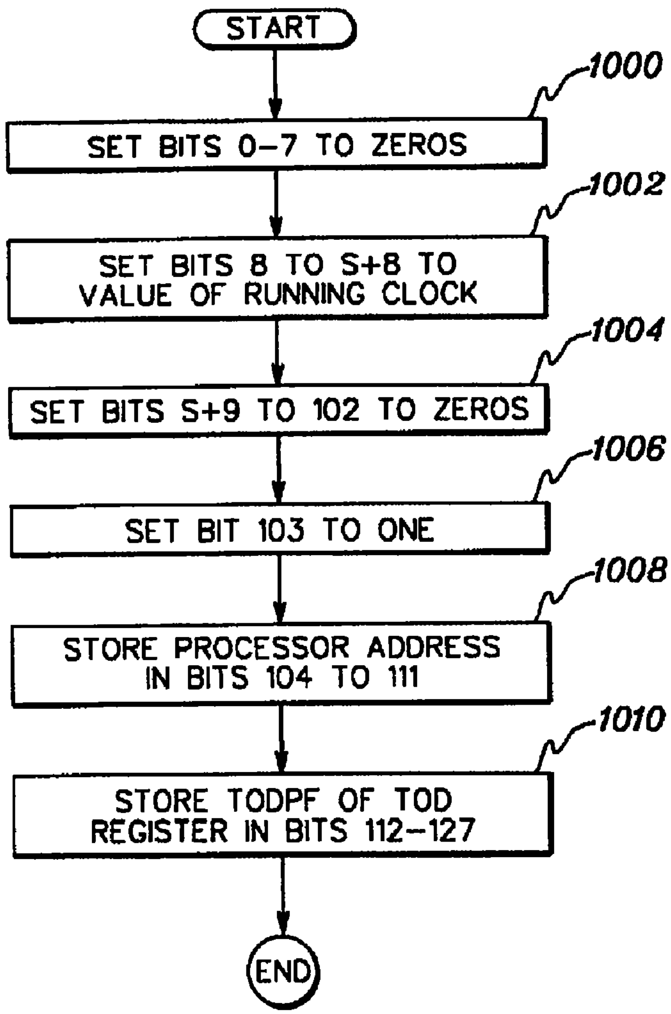

START
1000
SET BITS 0–7 TO ZEROS
1002
SET BITS 8 TO S+8 TO VALUE OF RUNNING CLOCK
1004
SET BITS S+9 TO 102 TO ZEROS
1006
SET BIT 103 TO ONE
1008
STORE PROCESSOR ADDRESS IN BITS 104 TO 111
1010
STORE TODPF OF TOD REGISTER IN BITS 112–127
END

fig. 10

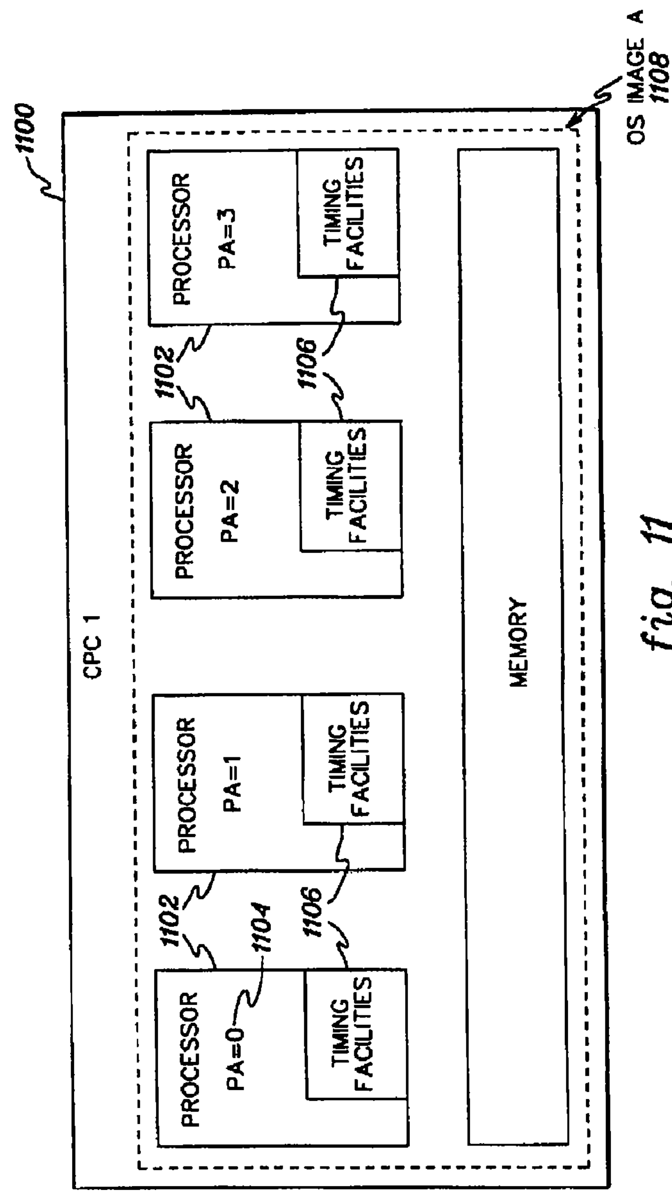

1100
CPC 1
OS IMAGE A
1108
PROCESSOR
PA=3
TIMING FACILITIES
PROCESSOR
PA=2
TIMING FACILITIES
PROCESSOR
PA=1
TIMING FACILITIES
PROCESSOR
PA=0
TIMING FACILITIES
1102
1104
1106
MEMORY

fig. 11

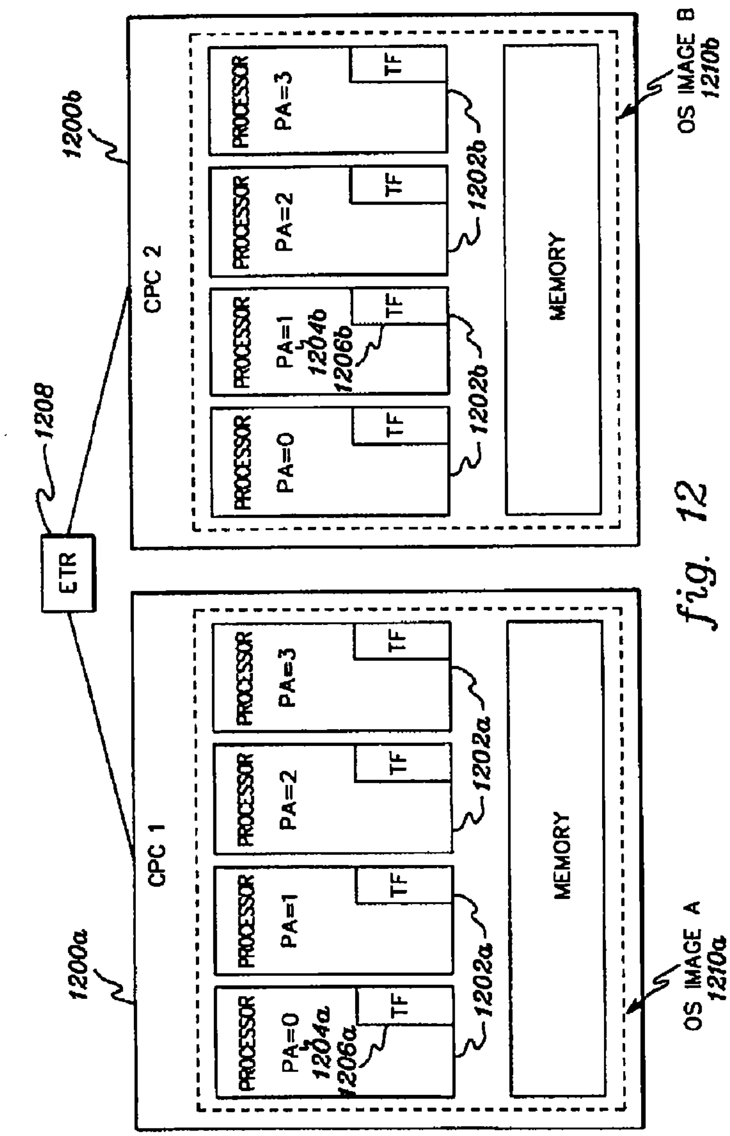

1200b
1208
ETR
CPC 2
PROCESSOR
PA=0
PA=1
PA=2
PA=3
TF
1204b
1206b
1202b
MEMORY
OS IMAGE B
1210b
1200a
CPC 1
1204a
1206a
1202a
OS IMAGE A
1210a

fig. 12

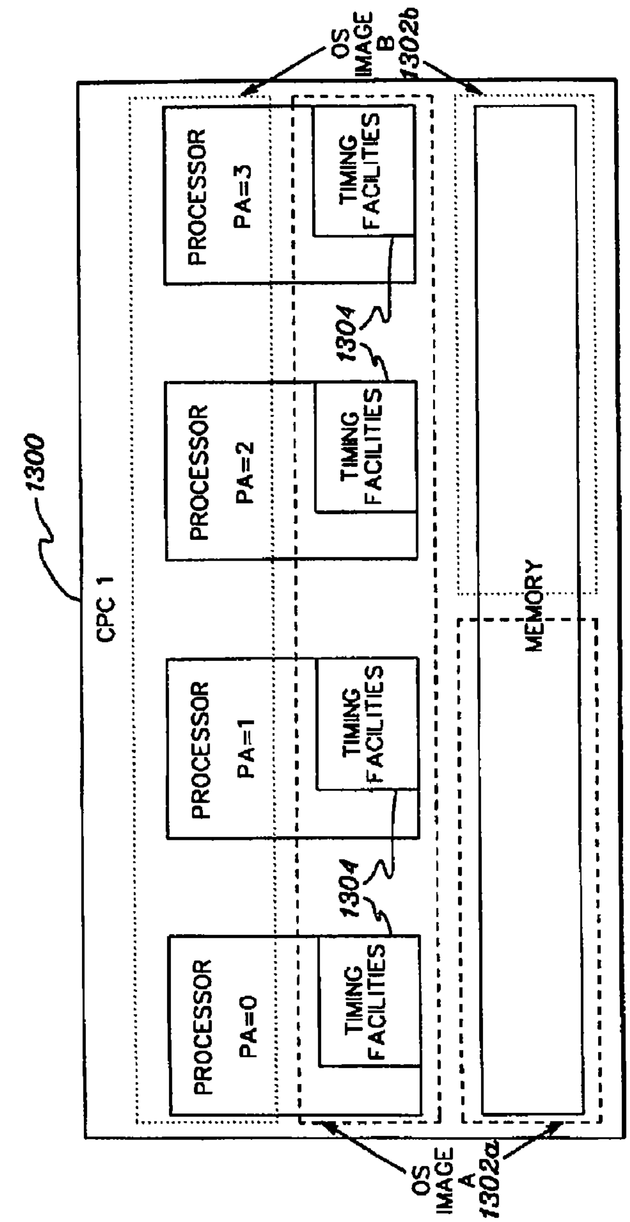

1300
CPC 1
PROCESSOR
PA=0
PROCESSOR
PA=1
PROCESSOR
PA=2
PROCESSOR
PA=3
TIMING FACILITIES
TIMING FACILITIES
TIMING FACILITIES
TIMING FACILITIES
1304
1304
MEMORY
OS IMAGE B
1302b
OS IMAGE A
1302a

fig. 13

METHOD, SYSTEM AND PROGRAM PRODUCTS FOR GENERATING SEQUENCE VALUES THAT ARE UNIQUE ACROSS OPERATING SYSTEM IMAGES

CROSS-REFERENCE TO RELATED APPLICATIONS

This application contains subject matter which is related to the subject matter of the following application, which is assigned to the same assignee as this application and filed on the same day as this application. The below listed application is hereby incorporated herein by reference in its entirety:

"METHOD, SYSTEM AND PROGRAM PRODUCTS FOR EMPLOYING EXPANDED PHYSICAL CLOCKS," by Elko et al., Ser. No. 09/337,157, (Docket No. PO9-99-064, now U.S. Pat. No. 6,490, 689), filed Jun. 21, 1999.

TECHNICAL FIELD

This invention relates, in general, to timing facilities within a computing environment and, in particular, to using the timing facilities to generate unique sequence values to be used by programs running within the computing environment.

BACKGROUND ART

Typically, processors of a computing environment either include or have access to timing facilities that provide date and time of day information. In the ESA/390 architecture offered by International Business Machines Corporation, the timing facilities include a time-of-day (TOD) clock, which provides a high-resolution measure of real-time suitable for the indication of the date and time.

In one example, the time-of-day clock is represented as a 64-bit-integer value that is set and incremented in an architecturally prescribed fashion based on real-time. This basic TOD clock is set to a value that corresponds to present time in Coordinated Universal Time (UTC), where bit 51 is updated once per microsecond and a clock value of zero corresponds to Jan. 1, 1900, 0 a.m.

The TOD-clock facility of ESA/390 is based on various architectural requirements, which are summarized below:

1. Uniqueness: Two executions of a STORE CLOCK instruction (used by a program to obtain the date and/or time), possibly on different central processing units (CPUs), are to store different values. Programs are to be able to rely on this uniqueness rule to produce unique identifiers for new object instances.
2. Monotonicity: The values stored by two STORE CLOCK instructions correctly imply the sequence of execution of the two instructions; namely, the instruction that occurs later in time stores a larger time value. This is true whether the two instructions are executed on the same or different CPUs. Programs are to be able to rely on this monotonicity rule to determine the sequence of occurrence of distinct events.
3. Predictable resolution: The resolution of the TOD clock is such that the incrementing rate is comparable to the instruction execution rate of the machine and should advance at least once during a time equal to, for instance, 10 average instructions. Performance characteristics of program loops can be determined by comparing time values at the beginning and end of the program. A predictable resolution allows these performance algorithms to be independent of the processor speeds.

Although efforts have been made to meet the above requirements, technological enhancements in processors have and continue to place strain on the ability to meet those requirements, as well as other requirements or features. Thus, enhanced timing facilities are needed to better meet the current requirements, as well as the requirements or desires of the future. For example, enhanced timing facilities are needed to meet the uniqueness requirement, when multiple STORE CLOCK instructions are executed on one or more CPUs having at least two operating system images.

SUMMARY OF THE INVENTION

The shortcomings of the prior art are overcome and additional advantages are provided through the provision of a method of generating unique sequence values usable within a computing environment. The method includes, for instance, providing as one part of a sequence value timing information including at least one of time-of-day information and date information; and including as another part of the sequence value selected information usable in making the sequence value unique across a plurality of operating system images on one or more processors of the computing environment.

As one example, the method also includes providing as a further part of the sequence value a placeholder value usable in ensuring that the sequence value is an increasing sequence value, even when a physical clock used to provide the timing information of the sequence value wraps back to zero.

In yet another embodiment of the present invention, the method includes providing as a further part of the sequence value a processor identifier.

In one example, the providing of the timing information and the including of the selected information are performed by an instruction.

In another embodiment of the present invention, a method of generating sequence values usable within a computing environment is provided. The method includes, for instance, providing as one part of a sequence value timing information including at least one of time-of-day information and date information; and including as another part of the sequence value placeholder information usable in ensuring that the sequence value is an increasing sequence value, even when a physical clock used to provide the timing information wraps back to zero.

In a further aspect of the present invention, a memory for storing data is provided, which includes a representation of a time-of-day clock. The representation is usable within a computing environment and includes, as one example, a timing component having timing information including at least one of time-of-day information and date information; and a programmable field component including selected information to provide from the representation a sequence value that is unique across a plurality of operating system images on one or more processors of the computing environment.

In yet another aspect of the present invention, a memory for storing data is provided, which includes a representation of a time-of-day clock. The representation is usable within a computing environment and includes, for instance, a timing component including timing information which includes at least one of time-of-day information and date information, the timing information being at least a part of a value resulting from the representation; and a placeholder component usable in ensuring that the value is an increasing sequence value, even when a physical clock used to provide the timing information wraps back to zero.

In another embodiment of the present invention, a system of generating unique sequence values usable within a computing environment is provided. The system includes, for instance, means for providing as one part of a sequence value timing information including at least one of time-of-day information and date information; and means for including as another part of the sequence value selected information usable in making the sequence value unique across a plurality of operating system images on one or more processors of the computing environment.

In another aspect of the present invention, a system of generating sequence values usable within a computing environment is provided. The system includes, for instance, means for providing as one part of a sequence value timing information including at least one of time-of-day information and date information; and means for including as another part of the sequence value placeholder information usable in ensuring that the sequence value is an increasing sequence value, even when a physical clock used to provide the timing information wraps back to zero.

In yet another aspect of the present invention, a system of generating unique sequence values usable within a computing environment is provided. The system includes, for instance, a processor adapted to provide as one part of a sequence value timing information including at least one of time-of-day information and date information; and the processor being further adapted to provide as another part of the sequence value selected information usable in making the sequence value unique across a plurality of operating system images on one or more processors of the computing environment.

In another aspect of the present invention, a system of generating sequence values usable within a computing environment is provided. The system includes a processor adapted to provide as one part of a sequence value timing information including at least one of time-of-day information and date information; and the processor being further adapted to provide as another part of the sequence value placeholder information usable in ensuring that the sequence value is an increasing sequence value, even when a physical clock used to provide the timing information wraps back to zero.

In another aspect of the present invention, an article of manufacture, including at least one computer usable medium having computer readable program code means embodied therein for causing the generating of unique sequence values usable within a computing environment is provided. The computer readable program code means in the article of manufacture includes, for instance, computer readable program code means for causing a computer to provide as one part of a sequence value timing information comprising at least one of time-of-day information and date information; and computer readable program code means for causing a computer to include as another part of the sequence value selected information usable in making the sequence value unique across a plurality of operating system images on one or more processors of the computing environment.

In yet a further aspect of the present invention, at least one program storage device readable by a machine, tangibly embodying at least one program of instructions executable by the machine to perform a method of generating sequence values usable within a computing environment is provided. The method includes, for instance, providing as one part of a sequence value timing information including at least one of time-of-day information and date information; and including as another part of the sequence value placeholder information usable in ensuring that the sequence value is an increasing sequence value, even when a physical clock used to provide the timing information wraps back to zero.

Advantageously, the present invention provides a mechanism for generating unique sequence values that are usable by programs running within a computer environment. The sequence values are generated using timing facilities and are unique across a plurality of operating system images of one or more processors of the computing environment. In particular, the present invention advantageously provides for an extended time-of-day clock representation that includes timing information, as well as a programable field that is used to make a value resulting from the representation unique across a plurality of operating system images.

Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention.

BRIEF DESCRIPTION OF THE DRAWINGS

The subject matter which is regarded as the invention is particularly pointed out and distinctly claimed in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages of the invention are apparent from the following detailed description taken in conjunction with the accompanying drawings in which:

FIG. 1 depicts one example of a computing environment incorporating and using the timing facilities of the present invention;

FIG. 2 is one example of a Symmetrical Multiprocessor (SMP) environment incorporating and using the timing facilities of the present invention;

FIG. 3 is one example of a Sysplex environment incorporating and using the timing facilities of the present invention;

FIG. 4 depicts one example of a representation of an extended time-of-day clock, in accordance with the principles of the present invention;

FIG. 5 depicts one embodiment of a representation of a basic time-of-day clock used in accordance with the principles of the present invention;

FIG. 6 depicts one embodiment of a time-of-day programmable register used in accordance with the principles of the present invention;

FIG. 7 depicts one example of a SET CLOCK PROGRAMMABLE FIELD instruction used in accordance with the principles of the present invention;

FIG. 8 depicts one examples of a STORE CLOCK EXTENDED instruction used in accordance with the principles of the present invention;

FIG. 9 depicts one embodiment of the logic associated with the STORE CLOCK EXTENDED instruction of FIG. 8, in accordance with the principles of the present invention;

FIG. 10 depicts one embodiment of the logic associated with constructing an extended time-of-day clock value, in accordance with the principles of the present invention; and

FIGS. 11–13 are further examples of computing environments incorporating and using the timing facilities of the present invention.

BEST MODE FOR CARRYING OUT THE INVENTION

In accordance with the principles of the present invention, timing facilities are provided that enhance the ability to

provide faster physical clocks, to provide unique clock values for multiple operating system images and to provide an ever-increasing sequence of numbers for the clock values.

The enhanced timing facilities include, for instance, an extended time-of-day clock representation, a programmable register and new instructions, which are described below.

The timing facilities of the present invention are included in, for example, a computing unit 100 (FIG. 1) based on the Enterprise Systems Architecture (ESA)/390 offered by International Business Machines Corporation, Armonk, N.Y. ESA/390 is described in an IBM Publication entitled "Enterprise Systems Architecture/390 Principles of Operation," IBM Publication No. SA22-7201-04, June 1997, which is hereby incorporated herein by reference in its entirety. In one embodiment, computing unit 100 is an ES/9000 computer, which includes at least one central processing unit 102, a main storage 104 and one or more input/output devices 106.

As one example, each central processing unit 102 includes timing facilities 108. However, in another embodiment, each central processing unit (or a subset thereof) is coupled to the timing facilities, which are located in a shared component, such as a shared memory bus.

Computing unit 100 may be a stand-alone computer or it may be included in a larger computing environment, such as, for example, a Symmetrical Multiprocessor (SMP) environment or a Sysplex environment offered by International Business Machines Corporation.

One example of an SMP environment is depicted in FIG. 2. SMP environment 200 includes a plurality of computing units 202 coupled to one another via, for example, a switch. The plurality of computing units are packaged in a frame 204, which includes, for instance, up to 16 computing units. In an SMP environment, all of the computing units share one operating system image. SMP is further described in "Enterprise Systems Architecture (ESA)/390 Principles Of Operation," IBM Publication No. SA22-7201-04, June 1997, which is hereby incorporated herein by reference in its entirety.

One embodiment of a Sysplex environment is described with reference to FIG. 3. A Sysplex environment 300 includes, for instance, one or more central processing complexes (CPCs) 302 coupled to at least one coupling facility 304.

Each central processing complex includes, for example, at least one central processing unit 306. Each central processing unit includes timing facilities 308 and executes an operating system 310, such as the Multiple Virtual Storage (MVS) or OS/390 operating system offered by International Business Machines Corporation. As one example, the same operating system image is executed by each central processing unit of CPC 1, while a different operating system image is executing on CPC 2 (e.g., OS/390 image A is executing on CPC 1 and OS/390 image B is executing on CPC 2; or OS/390 image A and another compatible operating system).

In another example, the operating system image executing on CPU 1 of CPC 1 is different from the operating system image executing on CPU 2 of CPC 1.

Each central processing complex 302 is coupled to coupling facility 304 (a.k.a., a structured external storage (SES) processor). Coupling facility 304 contains storage accessible by the central processing complexes and performs operations requested by programs in the CPCs. Aspects of the operation of a coupling facility are described in detail in such references as Elko et al., U.S. Pat. No. 5,317,739, entitled "Method And Apparatus For Coupling Data Processing Systems", issued May 31, 1994; Elko et al., U.S. Pat. No. 5,561,809, entitled "In a Multiprocessing System Having A Coupling Facility, Communicating Messages Between The Processors And The Coupling Facility In Either A Synchronous Operation or an Asynchronous Operation", issued on Oct. 1, 1996; Elko et al., U.S. Pat. No. 5,706,432, entitled "Mechanism For Receiving Messages At A Coupling Facility", issued Jan. 6, 1998, and the patents and applications referred to therein, all of which are hereby incorporated herein by reference in their entirety.

Although various computing environments are described above, those environments are only put forth as examples. The capabilities of the present invention can be used with other computing units, computing systems and or computing environments, without departing from the spirit of the present invention.

In one embodiment, the timing facilities of the present invention include a representation of an extended time-of-day (TOD) clock. One example of such a representation is described with reference to FIG. 4. An extended time-of-day clock representation 400 is, for instance, a 128-bit integer value that contains, for example, four components: a time-of-day epoch index (TEX) field 402, which is included in bits 0–7 of the extended time-of-day clock representation; a physical TOD clock field 404, which is included in bits 8 to s+8 and is referred to as the basic time-of-day clock; a processor identifier field 406, which is located in bits 104–111 of the extended time-of-day clock representation; and a TOD programmable field (TODPF) 408, which is contained in bits 112–127. Each of these fields is further described below.

TOD epoch index field (TEX) 402 is a one byte value, which is stored with a value of 0, at this time. This field is usable for further extensions of the time-of-day clock. For example, the bits of this field will be used to extend the physical TOD clock past Sep. 17, 2042 at 11:54 pm.

In particular, since one embodiment of the basic TOD clock (i.e., the physical clock or clock register) is established on an absolute time base, zero corresponds to Jan. 1, 1900 and bit 51 is defined as a microsecond increment. Thus, the basic clock will wrap back to zero at a prescribed time in the future (i.e., Sep. 17, 2042 at 11:54 pm.). Up until that point in time, the clock values represent an increasing sequence of numbers; a characteristic on which programs written to use the TOD clock rely.

In accordance with the principles of the present invention, the TOD epoch index adds, for example, eight additional bits to the left of bit 0 of the basic TOD clock value. Thus, values beyond 2043 will be handled by nonzero TEX values which will handle carries out of the basic TOD clock. This delays the wrapping problem for 36,534 years.

Programs that will continue to run on machines that support time values beyond the standard epoch will be written to allow for nonzero TEX values.

The TOD Epoch Index is, at this time, stored as zero in ESA/390 machines. However, future models will need to support time values that exceed the end of the standard epoch, which will occur in the year 2042. Testing requirements and machine longevity dictate that such '2042-capable' machines should be made available. The timing facility architecture for these machines are to be extended to support a nonzero TOD Epoch Index. This includes the capability to propagate carries from bit 8 to the extended form and to set the TOD-epoch index to any value when the clock is set.

The use of the TOD epoch index ensures that each value (e.g., the 128 bits) resulting from the extended time-of-day

clock representation is in increasing sequence order, even when the physical clock that provides the value for basic TOD field 404 wraps back to zero.

Basic TOD clock 404 is a representation of the physical clock used by the programs to obtain timing information (e.g., date information and/or time-of-day information). The physical clock is the running clock that is incremented on a predefined basis. The basic time-of-day clock representation is further described in "Enterprise Systems Architecture/390 Principles of Operation," IBM Publication No. SA22-7201-04, June 1997, which is hereby incorporated herein by reference in its entirety.

TOD clock 404 is also further described in more detail with reference to FIG. 5. In particular, bits 0–s of the representation of the basic time-of-day clock are shown in FIG. 5. In the basic form, the physical TOD clock is incremented by adding a 1 in bit position 51 every microsecond. In models having a higher or lower resolution, a different bit position, called the stepping bit (s) is incremented at such a frequency that the rate of advancing the clock is the same as if a 1 were added in bit position 51 in the basic form every microsecond. The stepping bit is the rightmost bit in the TOD clock that is incremented, and the frequency with which the stepping bit is incremented is called the incrementing rate.

Stepping bit 's' in the basic form corresponds to stepping bit 's+8' in the extended form (see FIG. 4), provided the bit position, s+8, is less than or equal to bit position 71 in the extended form. In this case, the resolution of the two forms of the clock are the same and bits 0 to s of the basic form and bits 8 to s+8 of the extended form are synchronized. If the stepping bit position s+8 corresponds to a bit to the right of bit 71 in the extended form, then the resolutions of the two forms differ. In this case, the stepping bit for the basic form is bit 63, which increments approximately once each 244 picoseconds, and bits 0 to 63 of the basic form and bits 8 to 71 of the extended form are synchronized. The resolution of the TOD clock is such that the incrementing rate is comparable to the instruction-execution rate of the model.

A TOD clock is said to be in a particular multiprocessing configuration if at least one of the CPUs which shares that clock is in the configuration. Conversely, if all CPUs having access to a particular TOD clock have been removed from a particular configuration, then the TOD clock is no longer considered to be in that configuration.

When more than one TOD clock exists in the configuration, the stepping rates are synchronized such that all TOD clocks in the configuration are incremented at the same rate.

When incrementing of the clock causes a carry to be propagated out of bit position 0 in the basic form, the carry is ignored, and counting continues from zero. In the extended form in which there is a placeholder, future models may support a carry provided by an extended physical clock.

Returning to FIG. 4, extended time-of-day clock representation 400 also includes processor identifier field 406. Processor identifier field 406 includes, for instance, the address of the processor executing a STORE CLOCK or STORE CLOCK EXTENDED instruction (described below) to obtain the timing information. This field is used to ensure the value obtained from the representation is unique across processors, which use a single operating system image. (In another embodiment, an identifier other than an address may be used.)

TOD Programmable Field (TODPF) 408 is, for instance, a 16-bit quantity that can be used for various purposes, including to provide system identifying information (e.g., an operating system identifier). This allows the uniqueness of the TOD clock values resulting from representation 400 to be extended to, for instance, different operating system images in a Sysplex environment.

TOD programmable field 408 corresponds to a time-of-day programmable field 602 (FIG. 6) of a TOD programmable register 600. In register 600, the TOD programmable field is, for instance, a 16-bit quantity contained in bit positions 16–31 of the TOD programmable register. Bits 0–15 of the register currently contain zeros and can be used for further expansion.

In one embodiment, a TOD programmable register exists for each CPU of the system and the contents of the TOD programmable register can be set by a privileged instruction, referred to as a SET CLOCK PROGRAMMABLE FIELD (SCKPF) instruction. This instruction is independent of an instruction used to initialize the physical clock (e.g., SET CLOCK), and thus, can be executed at a time apart from when the clock is initialized. The contents of the register are reset to a value of all zeros by an initial CPU reset.

As one example, a SET CLOCK PROGRAMMABLE FIELD instruction 700 (FIG. 7) has an "E" format denoting that the operation uses implied operands and that it has an extended op-code field 702, which identifies the operation to be performed. With this instruction, bits 16–31 of a general register (e.g., general register 0) are stored into the corresponding bit positions of TOD programmable register 600. Thus, when this instruction is executed, the identifier located in the general register is stored in the programmable register.

When there are multiple TOD programmable registers (e.g., in an SMP or a Sysplex where there is one register for each CPU), procedures are used to coordinate the values set in the registers. For each operating system image, the id of the image is obtained when the image first joins, e.g., a Sysplex. For each processor used to execute the image, the id is placed in its respective register.

The contents of register 600 and in particular, the contents of TODPF 602 of register 600, are stored within TOD programmable field 408 of the extended form TOD clock using, for instance, a STORE CLOCK EXTENDED instruction.

One example of a STORE CLOCK EXTENDED (STCKE) instruction 800 is described with reference to FIG. 8. STCKE instruction 800 has, for instance, an "S" format denoting an operation using an implied operand and storage. Instruction 800 includes an op code 802 specifying the operation to be performed and two storage operand fields 804, 806 to be used to determine the address in which the value of the extended TOD clock representation is to be stored once it is constructed. (As is known, B2 is a base register and D2 is the displacement to be added to the contents of B2 to form a second-operand address.)

The STORE CLOCK EXTENDED instruction provides unique sequence values from the extended TOD clock representation without requiring communication between different operating system images. Thus, the instruction is independent of such communication. One embodiment of the logic associated with executing the STORE CLOCK EXTENDED instruction is described with reference to FIG. 9.

Initially, when the STORE CLOCK EXTENDED instruction is executed by a program wishing to obtain the date and/or time of day, an extended time of day clock value is constructed by filling in representation 400, STEP 900. In one embodiment, the value is constructed, as descrited below with reference to FIG. 10.

In particular, bits 0–7 (i.e., the TEX) are set to zeros, STEP 1000. Further, bits 8 to s+8 are set to the value of the running physical clock, STEP 1002. In one example, the running clock is located in the CPU. In other embodiments, it is located elsewhere, such as within a shared memory bus.

Additionally, bits s+9 to 102 are set to zeros, STEP 1004, and bit 103 is set to one, STEP 1006. The setting of bit 103 to one ensures that the extended form time-of-day values are unique when compared with the basic form time-of-day values extended with zeros.

Further, the processor address (or another identifier) of the processor executing the instruction is placed in bits 104 to 111 of the extended time-of-day clock representation, STEP 1008. Additionally, the TODPF field of the TOD register is read and stored into bits 112–127 of the clock representation, STEP 1010.

Returning to FIG. 9, after constructing the extended time-of-day clock value, a determination is made as to which state the physical clock is in, INQUIRY 902. The clock may be in one of various states, including, for instance, stopped, set, not-set, error, and not-operational. The state determines the condition code set by execution of the STORE CLOCK instruction (used for basic TOD clocks) and the STORE CLOCK EXTENDED instruction (used for extended TOD clocks). Each of the states is described further below. The states are described with reference to the ESA/390 architecture.

Stopped State

The clock enters the stopped state when a SET CLOCK instruction is executed on a CPU accessing that clock and the clock is set. This occurs when SET CLOCK is executed without encountering any exceptions, and either (a) any manual TOD-clock control in the configuration is set to the enable-set position, or (b) the TOD-clock-control-override facility is installed and bit 10 of control register 14 is set to one. The clock can be placed in the stopped state from the set, not-set, and error state. The clock is not incremented while in the stopped state.

When the clock is in the stopped state, execution of the STORE CLOCK instruction or STORE CLOCK EXTENDED instruction on a CPU accessing that clock causes condition code 3 to be set and the value of the stopped clock to be stored.

One example of a SET CLOCK instruction is described in "Enterprise Systems Architecture/390 Principles of Operation", IBM Pub. No. SA22-7201-04, June 1997, which is hereby incorporated herein by reference in its entirety.

Set State

The clock enters the set state from the stopped state. The change of state is under control of a TOD-clock sync control bit, bit 2 of control register 0, in the CPU which most recently caused that clock to enter the stopped state. If the bit is zero, the clock enters the set state at the completion of execution of SET CLOCK. If the bit is one, the clock remains in the stopped state until (a) the bit is set to zero on that CPU; (b) another CPU executes a SET CLOCK instruction affecting the clock; (c) any other clock in the configuration is incremented to a value of all zeros in bit positions 32 through the rightmost bit position that is incremented when the clock is running or (d) with an external time reference (ETR), a signal from the ETR is used to set the set state. If any clock is set to a value of all zeros in bit positions 32 through the stepping bit and enters the set state as the result of a signal from another clock, or the ETR, the updating of bit positions 32 through the stepping bit of the two clocks is in synchronism.

Incrementing of the clock begins with the first stepping pulse after the clock enters the set state.

When the clock is in the set state, execution of the STORE CLOCK instruction or STORE CLOCK EXTENDED instruction causes condition code 0 to be set and the current value of the running clock to be stored.

Not-Set State

The clock is incremented, and is considered running, when it is in either the set state or the not-set state. When the power for the clock is turned on, the clock is set to zero, and the clock enters the not-set state. The clock is incremented when in the not-set state.

When the clock is in the not-set state, execution of the STORE CLOCK instruction or STORE CLOCK EXTENDED instruction causes condition code 1 to be set and the current value of the running clock to be stored.

Error State

The clock enters the error state when a malfunction is detected that is likely to have affected the validity of the clock value. A timing-facility-damage machine-check-interruption condition is generated on each CPU which has access to that clock whenever it enters the error state.

When STORE CLOCK or STORE CLOCK EXTENDED is executed and the clock accessed is in the error state, condition code 2 is set, and the value stored is zero.

Not-Operational State

The clock is in the not-operational state when its power is off or when it is disabled for maintenance. It depends on the model, if the clock can be placed in this state. Whenever the clock enters the not-operational state, a timing-facility-damage machine-check-interruption condition is generated on each CPU that has access to that clock.

When the clock is in the not-operational state, execution of STORE CLOCK or STORE CLOCK EXTENDED causes condition code 3 to be set, and zero is stored.

Continuing with FIG. 9, when the clock is in a set, stopped or not-set state, INQUIRY 902, the value constructed in STEP 900 is stored in the address designated by the second operand address so that the value is available to the program issuing the instruction, STEP 904. Thereafter, the STORE CLOCK EXTENDED instruction is complete, STEP 906.

Returning to INQUIRY 902, if the clock is not in the set, stopped or not-set state, but is in an error or not-operational state, then zeros are stored at the second operand address, STEP 910, and the instruction is complete, STEP 906.

(In another embodiment, the state of the clock is checked prior to constructing the extended time-of-day value such that the value is not constructed, when the clock is in the error or not-operational state.)

A serialization function is performed before the value of the clock is fetched and again after the value is placed in storage.

The quality of the clock value stored by the instruction is indicated by the resultant condition-code setting.

Resulting Condition Code:

0 Clock in set state

1 Clock in not-set state

2 Clock in error state

3 Clock in stopped state or not-operational state.

Condition code 0 normally indicates that the clock has been set by the control program. Accordingly, the value may be used in elapsed-time measurements and as a valid time-of-day and calendar indication. Condition code 1 indicates that the clock value is the elapsed time since the power for the clock was turned on. In this case, the value may be used in elapsed-time measurements, but is not a valid time-of-day indication. Condition codes 2 and 3 mean that the value provided by STORE CLOCK EXTENDED cannot be used for time measurement or indication.

Condition code 3 indicates that the clock is in either the stopped state or the not-operational state. These two states can normally be distinguished because the all-zero value is stored when the clock is in the not-operational state.

The physical clock can be set to a specific value by execution of SET CLOCK, if a manual TOD-clock control of any CPU in the configuration is in the enable-set position. Setting the clock replaces the values in the bit positions of the basic form from bit position 0 through the stepping bit when the clock is running (see FIG. 5). However, on some models, the rightmost bits starting at or to the right of bit 52 of the specified value are ignored, and zeros are placed in the corresponding positions of the clock. Zeros are also placed in positions to the right of bit position 63 of the clock.

The TOD clock can be inspected by executing STORE CLOCK, which causes a 64-bit value to be stored, or by executing STORE CLOCK EXTENDED, which causes a 128-bit value to be stored. Two executions of STORE CLOCK or STORE CLOCK EXTENDED possibly on different CPUs in the same configuration, store different values if the clock is running or, if separate clocks are accessed, both clocks are running and are synchronized.

The values stored for a running clock correctly imply the sequence of execution of any combination of STORE CLOCK and STORE CLOCK EXTENDED on one or more processors in the same configuration for all cases where the sequence can be established by means of the program. To ensure that unique values are obtained when the value of a running clock is stored, nonzero values may be stored in positions to the right of the stepping bit. When the value of a running clock is stored by means of STORE CLOCK EXTENDED, the value in bit positions 72–127 is nonzero. This ensures that the extended-form-TOD values are unique when compared with basic-form-TOD values extended with zeros.

For the purposes of uniqueness and sequence of execution between the results of STORE CLOCK and STORE CLOCK EXTENDED, the 64-bit value provided by STORE CLOCK may be considered to be extended to 128 bits by appending eight zeros on the left and 56 zeros on the right, then treating both values as 128-bit unsigned binary integers.

In a configuration where more than one CPU accesses the same clock, SET CLOCK is interlocked such that the entire contents appear to be updated concurrently; that is, if SET CLOCK instructions are executed simultaneously by two CPUs, the final result is either one or the other value. If SET CLOCK is executed on one CPU and STORE CLOCK or STORE CLOCK EXTENDED on the other, the result obtained by STORE CLOCK or STORE CLOCK EXTENDED is either the entire old value or the entire new value. When SET CLOCK is executed by one CPU, a STORE CLOCK or STORE CLOCK EXTENDED executed on another CPU may find the clock in the stopped state even when the TOD-clock-sync-control bit is zero in each CPU. (The TOD-clock-sync-control bit is bit 2 of control register 0.) Since the clock enters the set state before incrementing, the first STORE CLOCK or STORE CLOCK EXTENDED executed after the clock enters the set state may still find the original value introduced by SET CLOCK.

In accordance with the principles of the present invention, the uniqueness of the extended-form TOD clock value can be extended to separate processors in a multi-system installation by setting the TOD programmable field to a system-unique value on each processor in the configuration.

The uniqueness of the clock value on a single processor is ensured either by making the resolution of the clock sufficiently high in relation to the execution time of STORE CLOCK and STORE CLOCK EXTENDED or by interlocking the storing of the clock value such that execution of STORE CLOCK and STORE CLOCK EXTENDED are delayed until the clock has been incremented since the last execution of the instruction.

The uniqueness of the clock value on separate processors in the same configuration is ensured by placing a processor unique value in the rightmost bits of the clock. In previous models, this is accomplished by placing a four-bit processor address (PA) in bits 60–63 of the basic form TOD value. When the extended TOD clock facility is installed, the PA field should be placed to the right of bit 63, preferably in bits 108–111 of the extended form TOD value.

The uniqueness of the clock value between separate executions of STORE CLOCK and STORE CLOCK EXTENDED can be accomplished by selecting a b t in the range of bits 71 to 107 that is to the right of the stepping bit and setting the bit to one when the extended form is stored. This ensures that when zeros are appended to the basic form for the purposes of comparing with the extended form, the two resulting 128-bit integers are different. The recommended bit is bit 103. This is sufficiently far to right of the stepping bit that it should not need to be changed and it allows expansion of the PA field to the left by 4 bits, which would support 256 processors.

Correct sequencing of STORE CLOCK and STORE CLOCK EXTENDED can be accomplished on future machines by, for instance, delaying the execution of STORE CLOCK until bits 68–71 of the extended form TOD clock match the processor address on this CPU, followed by enough additional delay before storing to ensure that bits 8–71 of the extended form TOD clock value, as observed by all other CPUs in the configuration, is larger than the STORE CLOCK value stored by this CPU.

Advantageously, the extended TOD-clock facility of the present invention addresses various problems of the basic TOD clock by adding two new objects in the timing facilities: an extended form TOD clock and a TOD programmable register. Specifically, the various problems are addressed as follows:

1. Wrapping of the clock: Since the basic TOD clock is established on an absolute time base, zero corresponds to Jan. 1, 1900 and bit 51 is defined as a microsecond increment. Thus, the clock will wrap back to zero at a prescribed time in the future, i.e., Sep. 17, 2042 at 11:54 pm. Up until that point in time, the basic clock values represent an increasing sequence of numbers, a characteristic on which programs written to use the TOD clock rely.

   The extended TOD clock includes the TOD Epoch Index, which adds eight additional bits to the left of bit 0 of the basic S/390 TOD clock value. Thus, time values beyond 2042 are handled by nonzero TEX values, which handles carries out of the basic TOD. The existence of the TEX field delays the wrapping problem for 36,534 years. This design provides a smooth transition, creating a structure to support the additional 8 high-order bits, while not requiring that they be physically implemented immediately.

   The TOD Epoch index is a placeholder that ensures that the values resulting from the extended TOD clock representation are increasing sequence values, even when the physical clock wraps back to zero.

2. Precision limits: In the basic TOD clock, the right-most bit available for the stepping bit is bit 59, which corresponds with a clock precision of between 3 and 4

nanoseconds. Machine cycle times are already being pushed below this level, and in a very few machine generations, this limitation will affect the ability of the machine to implement a sufficiently fast physical clock that can also satisfy the uniqueness requirements.

The extended TOD clock moves the processor identifier past bit 104, allowing the precision of the physical clock to move an additional 36 bits to the right.

Additionally, in accordance with the principles of the present invention, the basic STORE CLOCK instruction can continue to be used, even after the precision of the physical clock is enhanced by adding one or more bits to the clock. In particular, the execution of the STORE CLOCK instruction is slowed down, so that clock values reflect the same bit settings on bits 60–63 on a given processor. That is, the STORE CLOCK instruction begins executing and places a value of the physical clock in the basic TOD clock field of a time-of-day clock representation. Since the time-of-day clock representation has not been expanded, the basic time-of-day clock field is unable to accommodate the value provided by the expanded physical clock. Thus, the value encroaches upon, at the very least, the processor identifier field of the basic time-of-day clock representation. Thus, the completion of the instruction is delayed until the processor identifier field reflects the correct processor identifier. As is known, the instruction is slowed down by having the microcode wait to issue an endop, which would then return control to the next instruction.

One example of the above is as follows:

Upon execution of a STORE CLOCK instruction, a value of the physical clock is stored in the representation, but the value encroaches upon the processor identifier field such that the processor identifier field now has, for example, bits 0001 located therein. However, the proper processor identifier is 1001. Thus, execution of the instruction is delayed until the bits that encroach upon the processor identifier are the same as the processor identifier (e.g., 1001).

3. SMP Extensions: Previously, the processor identifier field (or PA, in this example) allowed for 16 unique processors in, for instance, an SMP environment. However, advances in technology, packaging and memory design allow for more than 16 processors to be packaged in an SMP frame. Maintaining the uniqueness requirement requires the PA field be extended. Thus, with the extended time-of-day clock, the processor identifier field (e.g., processor address) is increased from, for example, 4 to 8 bits, allowing for systems to contain up to 256 processors.

4. Parallel Sysplex Extensions: When uniprocessor systems evolved into multiprocessor systems, the architecture evolved as well, enabling applications to develop multitasking equivalents. The basic TOD-clock architecture and the rules defined above were part of that evolution. Now that parallel sysplex has been introduced, it is important to evolve the architecture to allow applications to develop parallel equivalents. For the TOD-clock architecture that would mean, ideally, to extend the architecture rules to the sysplex environment.

The inclusion of the TOD Programmable field in the last 16 bits of the extended TOD clock allows the operating system to provide a system identifier in the TOD programmable register that would be stored by the new STORE CLOCK EXTENDED instruction, and thus, extend the uniqueness of the TOD clock values to different systems in a Parallel Sysplex.

ETR continues to be used as a synchronizing mechanism for the separate physical TOD clocks. ETR is described in "Sysplex Timer Planning," IBM Publication No. GA23-0365-02, 1993, which is hereby incorporated herein by reference in its entirety.

Further benefits of the present invention are explained with reference to FIGS. 11–13. FIG. 11 depicts one example of a central processing complex 1100 having four processors 1102 (i.e., central processing units) associated therewith. Each processor has its own processor identifier 1104 (e.g., processor address (PA)) and timing facilities 1106. The timing facilities in each processor are synchronized by hardware controls in the central processing complex.

This embodiment of a central processing complex is referred to as a 4 processor SMP. This 4 processor SMP has one copy of an operating system 1108 (i.e., Operating System Image A). That is, each of the four SMP processors shares the same operating system image. Unique values for the TOD clock are provided by the different processor identifier values, since there is only one operating system image.

FIG. 12 depicts two separate central processing complexes 1200a, 1200b, each having for example, four processors 1202a, 1202b, respectively. Each processor 1202a has a processor address 1204a and timing facilities 1206a. Likewise, each processor 1202b has a processor address 1204b and timing facilities 1206b. The timing facilities are synchronized within the central processing complexes by hardware controls in each central processing complex, and the timing facilities are synchronized between the central processing complexes by ETR connections 1208.

In this example, each central processing complex has a single copy of an operating system image 1210a, 1210b, respectively. For instance, Central Processing Complex 1 has an Operating System Image A and Central Processing Complex 2 has an Operating System Image B. Unique values for the TOD clock are provided by the programmable field, which provides, for instance, an identifier associated with the operating system, as well as by the processor identifier field, which provides uniqueness within an operating system image.

FIG. 13 depicts a single 4 processor SMP (CPC 1) 1300 that has been logically partitioned and is running two operating system images, Operating System Image A 1302a and Operating System Image B 1302b. The timing facilities 1304 are logically separated by an LPAR hipervisor and may be set to different values or coordinated depending on the configuration parameters. (LPAR is further described in "ES/9000 and ES/390 PR/SM Planning Guide," IBM Publication No. GA22-7123-13, March 1996, which is hereby incorporated herein by reference in its entirety. Uniqueness of the TOD values for STORE CLOCK instructions executed by separate tasks running on the same physical processor are not guaranteed by the processor identifier value, which will be the same. Thus, the time-of-day clock programmable field and the extended time-of-day clock representation are used to guarantee uniqueness.

The present invention can be included in an article of manufacture (e.g., one or more computer program products) having, for instance, computer usable media. The media has embodied therein, for instance, computer readable program code means for providing and facilitating the capabilities of the present invention. The article of manufacture can be included as a part of a computer system or sold separately.

Additionally, at least one program storage device readable by a machine, tangibly embodying at least one program of instructions executable by the machine to perform the capabilities of the present invention can be provided.

The flow diagrams depicted herein are just exemplary. There may be many variations to these diagrams or the steps (or operations) described therein without departing from the spirit of the invention. For instance, the steps may be performed in a differing order, or steps may be added, deleted or modified. All of these variations are considered a part of the claimed invention.

Although preferred embodiments have been depicted and described in detail herein, it will be apparent to those skilled in the relevant art that various modifications, additions, substitutions and the like can be made without departing from the spirit of the invention and these are therefore considered to be within the scope of the invention as defined in the following claims.

What is claimed is:

1. A method of generating unique sequence values usable within a computing environment, said method comprising:
   providing as one part of a sequence value timing information comprising at least one of time-of-day information and date information; and
   including as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment, wherein said sequence value is usable as a current time of day clock value in real-time processing by one or more processors of the computing environment.

2. The method of claim 1, further comprising providing as a further part of said sequence value a placeholder value usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information of said sequence value wraps back to zero.

3. The method of claim 1, further comprising providing as a further part of said sequence value a processor identifier.

4. The method of claim 1, wherein said providing and said including are performed by an instruction.

5. The method of claim 4, wherein said providing comprises retrieving, by said instruction, said timing information from a physical clock, and wherein said including comprises retrieving, by said instruction, said selected information from a storage area.

6. The method of claim 5, wherein said storage area is a programmable register set by a set register instruction.

7. The method of claim 6, further comprising initializing said physical clock independently of setting said programmable register.

8. The method of claim 4, wherein said instruction is a STORE CLOCK EXTENDED instruction.

9. The method of claim 4, wherein said instruction is issued by a program of said computing environment desiring said sequence value, and wherein said instruction is independent such that communication between said plurality of operating system images is not necessary to generate said sequence value.

10. The method of claim 1, further comprising receiving said timing information from a physical clock of said computing environment.

11. The method of claim 10, further comprising initializing said physical clock to a predefined value using a set instruction.

12. The method of claim 11, further comprising obtaining said selected information from a programmable register independent from said physical clock and said set instruction.

13. The method of claim 1, further comprising obtaining said selected information from a programmable register set by a set register instruction.

14. The method of claim 1, wherein the sequence value that is generated is considered as one entity, said one entity being usable as the current time of day clock value.

15. The method of claim 1, wherein said sequence value indicates a correct sequence of execution of an instruction, regardless of which processor of the one or more processors executes the instruction.

16. The method of claim 1, wherein the sequence value has a predictable resolution.

17. A memory for storing data, said memory comprising:
   a representation of a time-of-day clock, said representation being usable within a computing environment and providing a value usable as a current time of day clock value in real-time processing by one or more processors of the computing environment, said representation comprising:
      a timing component comprising timing information including at least one of time-of-day information and date information; and
      a programmable field component comprising selected information to make the value unique across a plurality of operating system images on one or more processors of said computing environment.

18. The memory of claim 17, wherein said representation further comprises a placeholder component usable in ensuring that said value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

19. The memory of claim 17, wherein said representation further comprises a processor identifier component including processor information.

20. A system of generating unique sequence values usable within a computing environment, said system comprising:
   means for providing as one part of a sequence value timing information comprising at least one of time-of-day information and date information;and
   means for including as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment, wherein said sequence value is usable as a current time of day clock value in real-time processing by one or more processors of the computing environment.

21. The system of claim 20, further comprising means for providing as a further part of said sequence value a placeholder value usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information of said sequence value wraps back to zero.

22. The system of claim 20, further comprising means for providing as a further part of said sequence value a processor identifier.

23. The system of claim 20, wherein said means for providing and said means for including comprise an instruction.

24. The system of claim 23, wherein said means for providing comprises means for retrieving, by said instruction, said timing information from a physical clock, and wherein said means for including comprises means for retrieving, by said instruction, said selected information from a storage area.

25. The system of claim 24, wherein said storage area is a programmable register set by a set register instruction.

26. The system of claim 25, further comprising means for initializing said physical clock independently of setting said programmable register.

27. The system of claim 23, wherein said instruction is issued by a program of said computing environment desiring said sequence value, and wherein said instruction is independent such that communication between said plurality of operating system images is not necessary to generate said sequence value.

28. The system of claim 20, further comprising means for receiving said timing information from a physical clock of said computing environment.

29. The system of claim 28, further comprising means for initializing said physical clock to a predefined value using a set instruction.

30. The system of claim 29, further comprising means for obtaining said selected information from a programmable register independent from said physical clock and said set instruction.

31. The system of claim 20, further comprising means for obtaining said selected information from a programmable register set by a set register instruction.

32. A system of generating unique sequence values usable within a computing environment, said system comprising:

a processor adapted to provide as one part of a sequence value timing information comprising at least one of time-of-day information and date information;and

said processor being further adapted to provide as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment, wherein said sequence value is usable as a current time of day clock value in real-time processing by one or more processors of the computing environment.

33. An article of manufacture, comprising:

at least one computer usable medium having computer readable program code means embodied therein for causing the generating of unique sequence values usable within a computing environment, the computer readable program code means in said article of manufacture comprising:

computer readable program code means for causing a computer to provide as one part of a sequence value timing information comprising at least one of time-of-day information and date information;and

computer readable program code means for causing a computer to include as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment, wherein said sequence value is usable as a current time of day clock value in real-time processing by one or more processors of the computing environment.

34. The article of manufacture of claim 33, further comprising computer readable program code means for causing a computer to provide as a further part of said sequence value a placeholder value usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information of said sequence value wraps back to zero.

35. The article of manufacture of claim 33, further comprising computer readable program code means for causing a computer to provide as a further part of said sequence value a processor identifier.

36. The article of manufacture of claim 33, wherein said computer readable program code means for causing a computer to provide comprises computer readable program code means for causing a computer to retrieve, by an instruction, said timing information from a physical clock, and wherein said computer readable program code means for causing a computer to include comprises computer readable program code means for causing a computer to retrieve, by said instruction, said selected information from a storage area.

37. The article of manufacture of claim 36, wherein said storage area is a programmable register set by a set register instruction.

38. The article of manufacture of claim 37, further comprising computer readable program code means for causing a computer to initialize said physical clock independently of setting said programmable register.

39. The article of manufacture of claim 33, further comprising computer readable program code means for causing a computer to receive said timing information from a physical clock of said computing environment.

40. The article of manufacture of claim 39, further comprising computer readable program code means for causing a computer to initialize said physical clock to a predefined value using a set instruction.

41. The article of manufacture of claim 40, further comprising computer readable program code means for causing a computer to obtain said selected information from a programmable register independent from said physical clock and said set instruction.

* * * * *

EXHIBIT 15

US005953520A

AA9-96-003

United States Patent [19]

Mallick

[11] Patent Number: 5,953,520

[45] Date of Patent: Sep. 14, 1999

[54] ADDRESS TRANSLATION BUFFER FOR DATA PROCESSING SYSTEM EMULATION MODE

[75] Inventor: Soummya Mallick, Austin, Tex.

[73] Assignee: International Business Machines Corporation, Armonk, N.Y.

[21] Appl. No.: 08/934,645

[22] Filed: Sep. 22, 1997

[51] Int. Cl.[6] G06F 9/455

[52] U.S. Cl. 395/500.47

[58] Field of Search 395/568, 385, 395/500

[56] References Cited

U.S. PATENT DOCUMENTS

5,339,417	8/1994	Connell et al.	395/650
5,574,873	11/1996	Davidian	395/376
5,742,802	4/1998	Harter et al.	395/568
5,790,825	11/1995	Traut	395/385

Primary Examiner—Kevin J. Teska

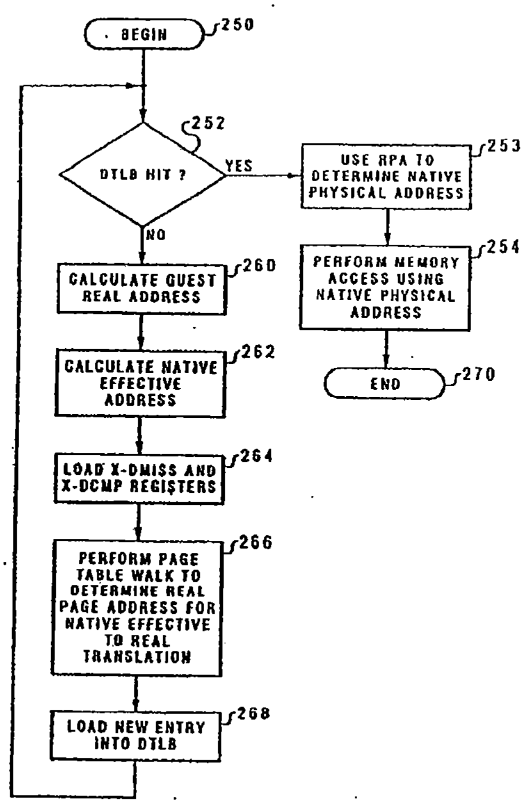
Assistant Examiner—Ayni Mohamed

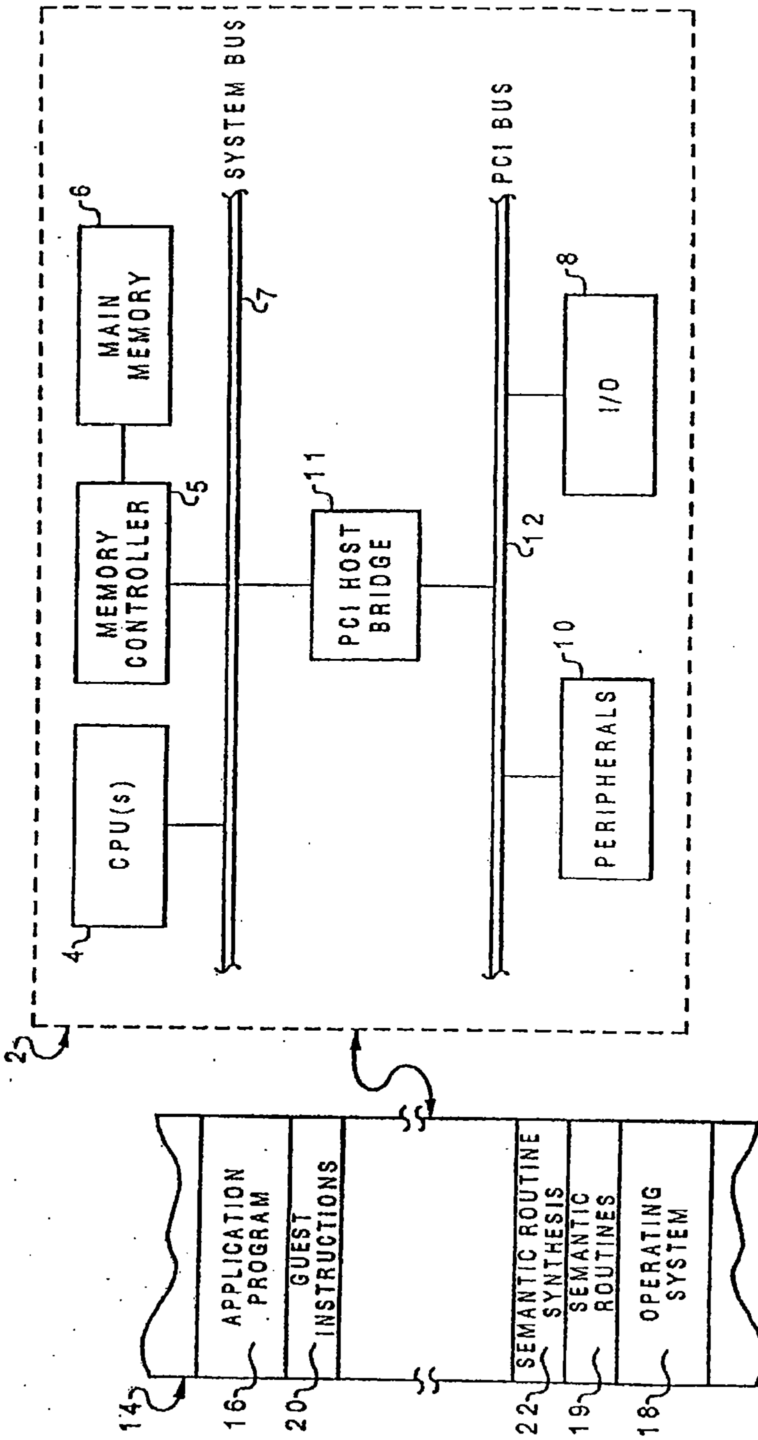
Attorney, Agent, or Firm—Casimer K. Salys; Brian F. Russell; Andrew J. Dillon

[57] ABSTRACT

A processor and method of operating a processor which has a native instruction set and emulates instructions in a guest instruction set are described. According to the method, a series of guest instructions from the guest instruction set are stored in memory. The series includes a guest memory access instruction that indicates a guest logical address in guest address space. For each guest instruction in the series, a semantic routine of native instructions from the native instruction set is stored in memory. The semantic routines, which utilize native addresses in native address space, can be executed in order to emulate the guest instructions. In response to receipt of the guest memory access instruction for emulation, the guest logical address is translated into a guest real address, which is thereafter translated into a native physical address. A semantic routine that emulates the guest memory access instruction is then executed utilizing the native physical address.

16 Claims, 9 Drawing Sheets





MAIN MEMORY
6
MEMORY CONTROLLER
5
CPU(s)
4
SYSTEM BUS
7
PCI HOST BRIDGE
11
PCI BUS
12
I/O
8
PERIPHERALS
10
2
APPLICATION PROGRAM
GUEST INSTRUCTIONS
SEMANTIC ROUTINE SYNTHESIS
SEMANTIC ROUTINES
OPERATING SYSTEM
14
16
20
22
19
18

Fig. 1

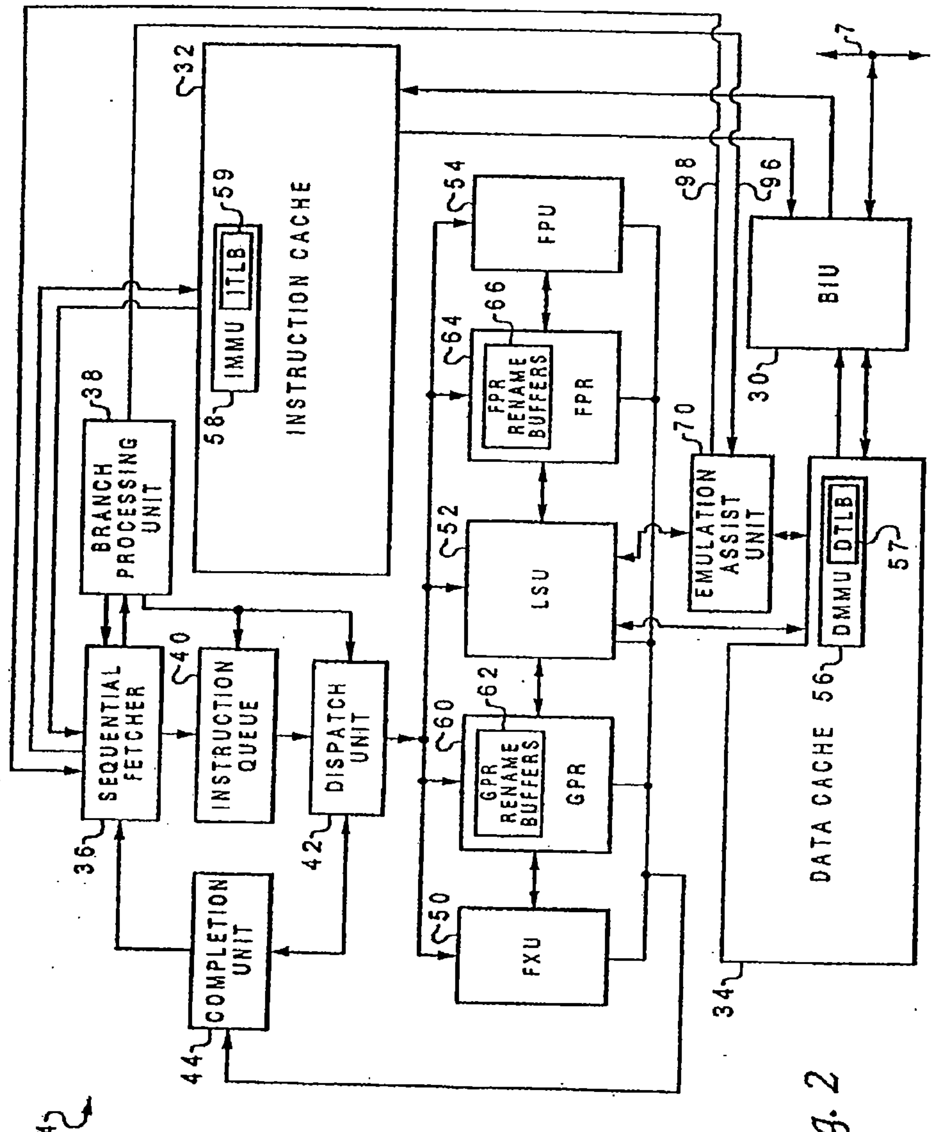

INSTRUCTION CACHE
IMMU
ITLB
BRANCH PROCESSING UNIT
SEQUENTIAL FETCHER
INSTRUCTION QUEUE
DISPATCH UNIT
COMPLETION UNIT
FPU
FPR RENAME BUFFERS
FPR
LSU
GPR RENAME BUFFERS
GPR
FXU
EMULATION ASSIST UNIT
BIU
DMMU
DTLB
DATA CACHE
4
7
30
32
34
36
38
40
42
44
50
52
54
56
57
58
59
60
62
64
66
70
96
98

Fig. 2

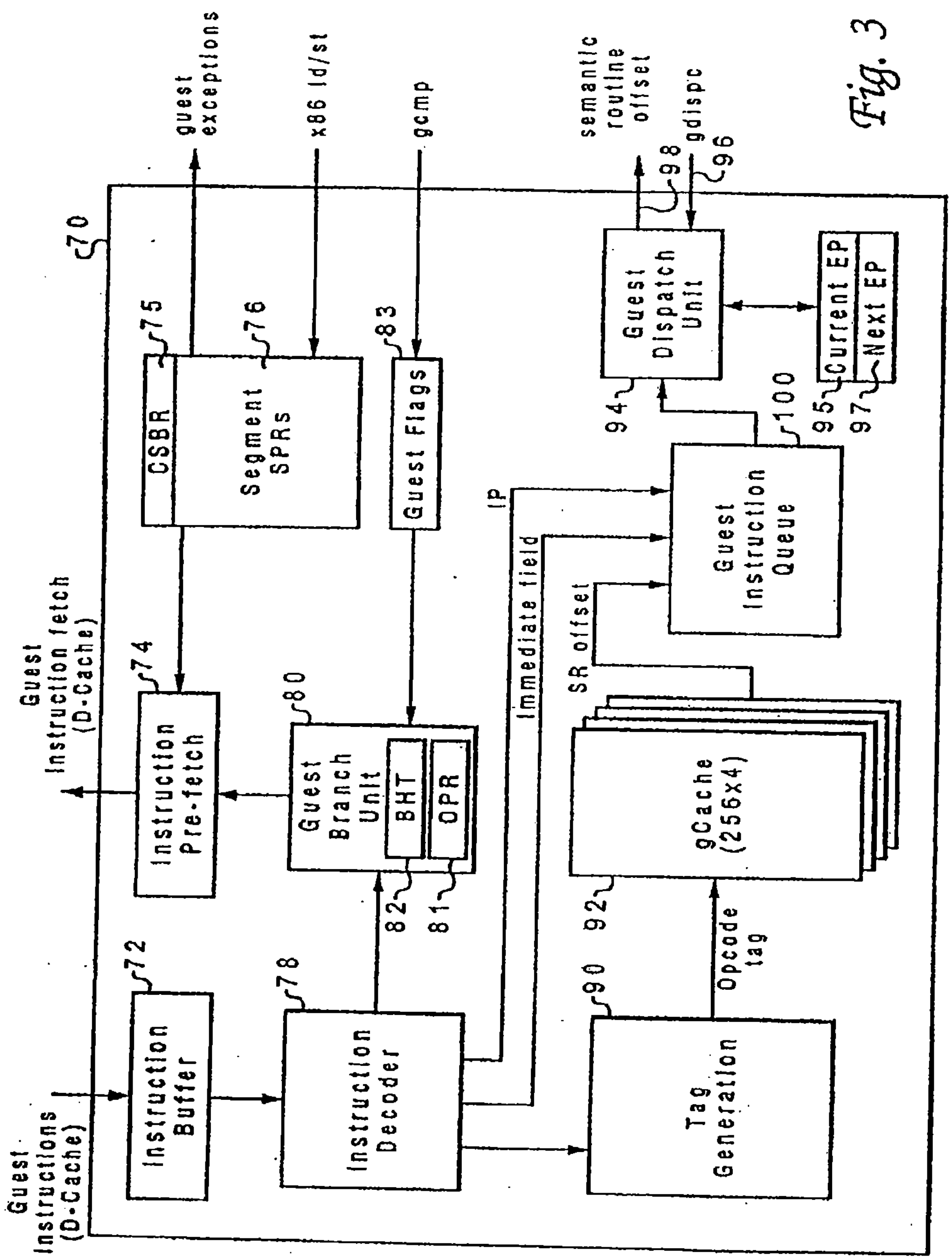

guest exceptions
x86 ld/st
gcmp
semantic routine offset
gdispc
70
75
CSBR
76
Segment SPRs
83
Guest Flags
Guest Dispatch Unit
98
96
94
Current EP
Next EP
95
97
100
IP
Immediate field
SR offset
Guest Instruction Queue
Guest Instruction fetch (D-Cache)
74
Instruction Pre-fetch
80
Guest Branch Unit
BHT
OPR
82
81
gCache (256x4)
92
Opcode tag
90
Tag Generation
72
Instruction Buffer
78
Instruction Decoder
Guest Instructions (D-Cache)

Fig. 3

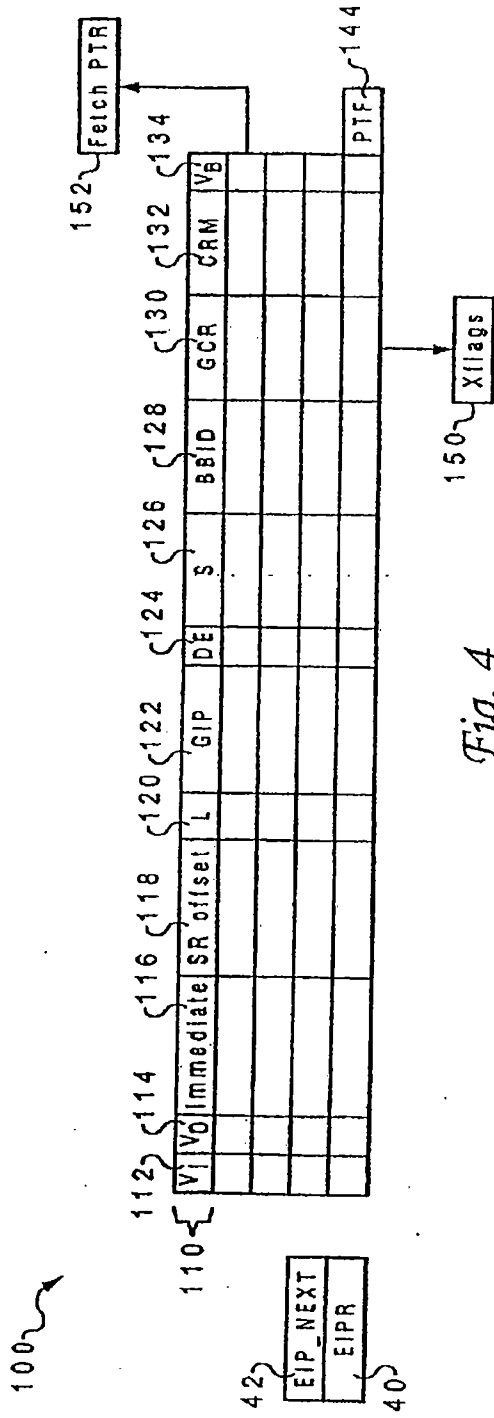

Fetch PTR
PTF
Xflags
EIP_NEXT
EIPR
Vi
Vo
Immediate
SR
offset
L
GIP
DE
S
BBID
GCR
CRM
VB
100
110
112
114
116
118
120
122
124
126
128
130
132
134
140
142
144
150
152

Fig. 4

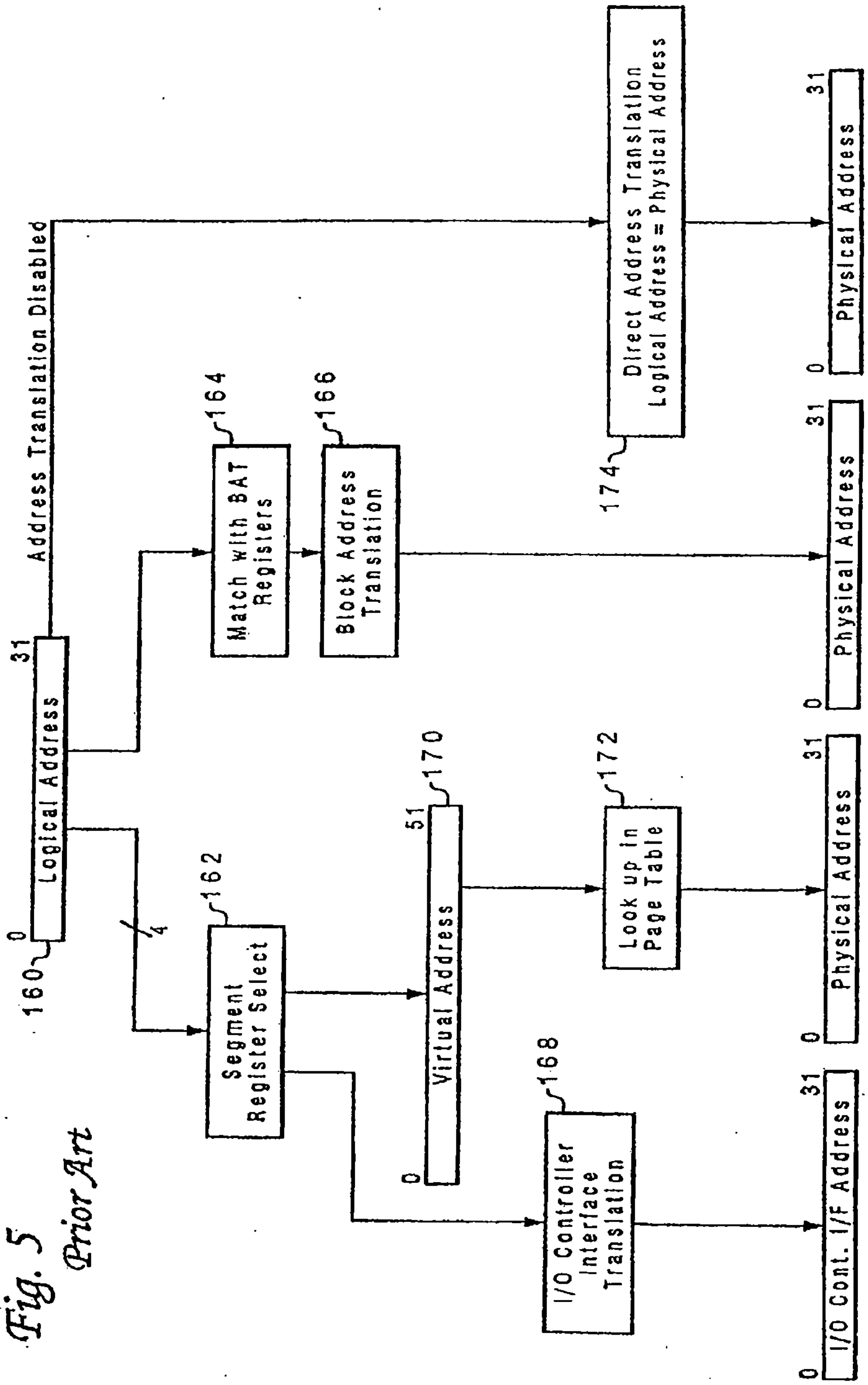


Fig. 5
Prior Art

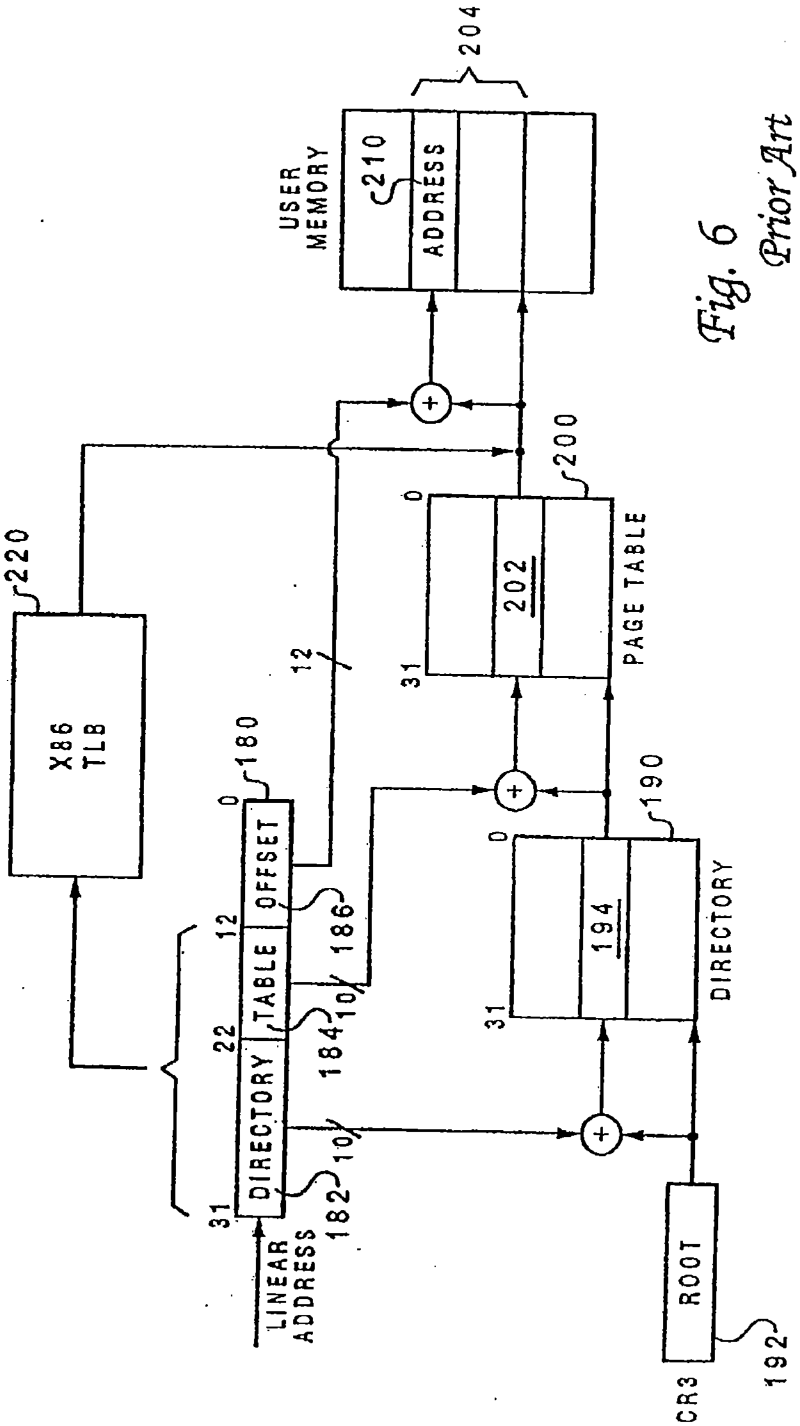
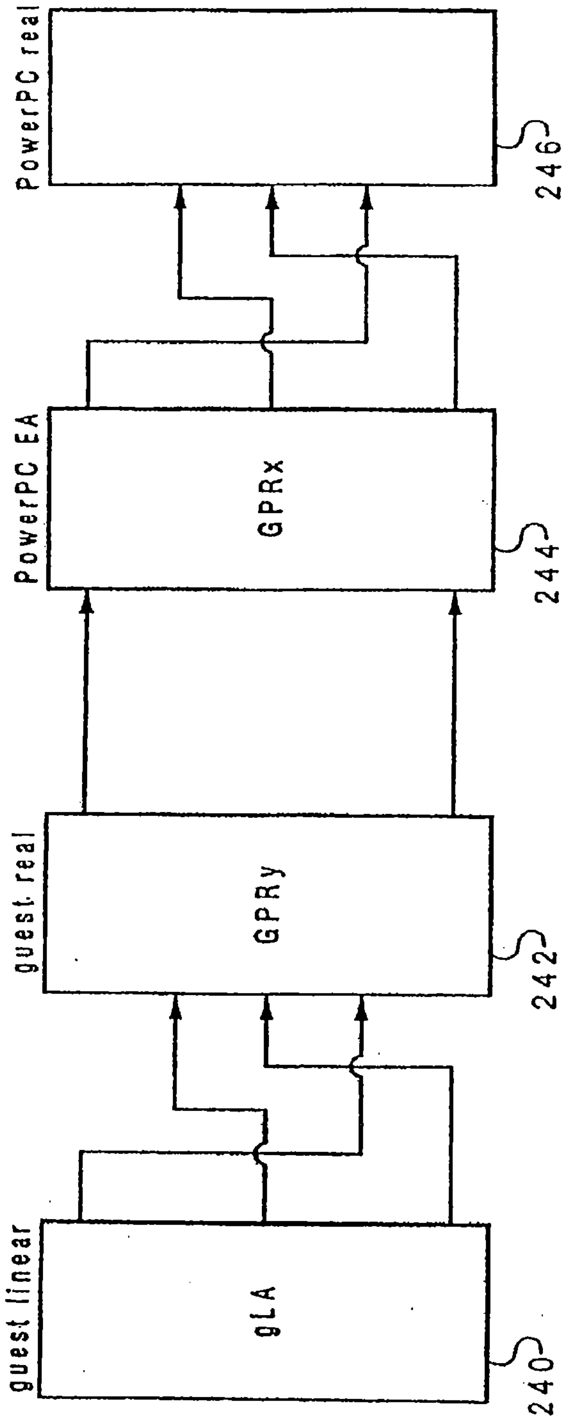
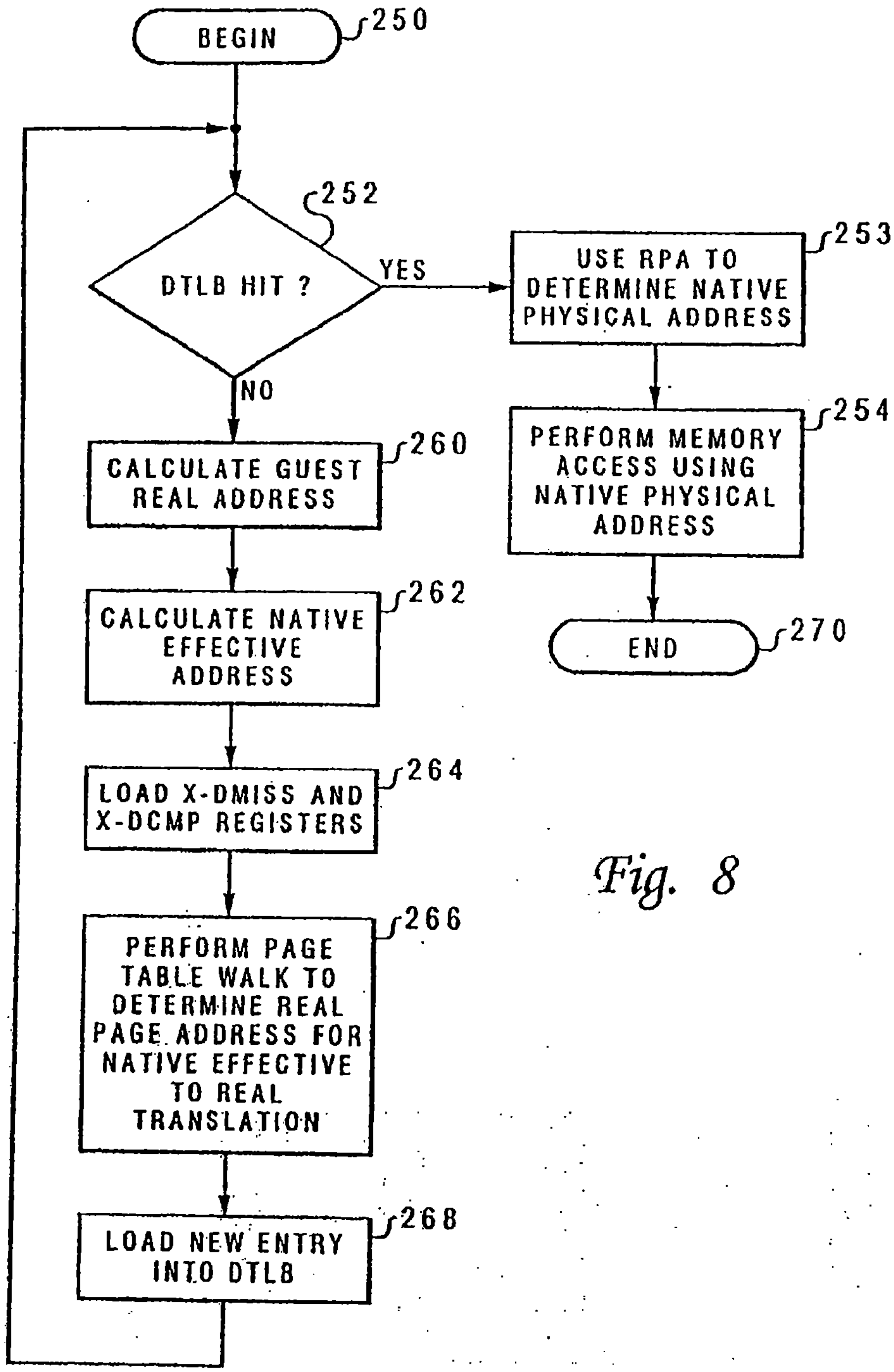


Fig. 6
Prior Art



PowerPC real
246
PowerPC EA
GPRx
244
guest real
GPRy
242
guest linear
gLA
240

Fig. 7



BEGIN
250
252
DTLB HIT ?
YES
NO
USE RPA TO DETERMINE NATIVE PHYSICAL ADDRESS
253
PERFORM MEMORY ACCESS USING NATIVE PHYSICAL ADDRESS
254
END
270
CALCULATE GUEST REAL ADDRESS
260
CALCULATE NATIVE EFFECTIVE ADDRESS
262
LOAD X-DMISS AND X-DCMP REGISTERS
264
PERFORM PAGE TABLE WALK TO DETERMINE REAL PAGE ADDRESS FOR NATIVE EFFECTIVE TO REAL TRANSLATION
266
LOAD NEW ENTRY INTO DTLB
268

Fig. 8

ADDRESS TRANSLATION BUFFER FOR DATA PROCESSING SYSTEM EMULATION MODE

CROSS-REFERENCE TO RELATED APPLICATIONS

The subject matter of this application is related to that disclosed in the following applications, which are assigned to the assignee of the present application and are incorporated herein by reference:

Ser. No. (08,934,644) now U.S. Pat. No. 5,870,575, filed of even date herewith, for INDIRECT UNCONDITIONAL BRANCHES IN DATA PROCESSING SYSTEM EMULATION MODE, by James A. Kahle and Soummya Mallick.

Ser. No. 08,934,857, filed of even date herewith, for METHOD AND SYSTEM FOR PROCESSING BRANCH INSTRUCTIONS DURING EMULATION IN A DATA PROCESSING SYSTEM, by James A. Kahle and Soummya Mallick.

Ser. No. 08,935,007, filed of even date herewith, for METHOD AND SYSTEM FOR INTERRUPT HANDLING DURING EMULATION IN A DATA PROCESSING SYSTEM, by James A. Kahle and Soummya Mallick.

Ser. No. 08/591,291, filed Jan. 25, 1996, for A METHOD AND SYSTEM FOR MINIMIZING THE NUMBER OF CYCLES REQUIRED TO EXECUTE SEMANTIC ROUTINES, by Soummya Mallick.

Ser. No. 08/581,793, filed Jan. 25, 1996, for A METHOD AND SYSTEM FOR IMPROVING EMULATION PERFORMANCE BY PROVIDING INSTRUCTIONS THAT OPERATE ON SPECIAL-PURPOSE REGISTER CONTENTS, by Soummya Mallick.

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates in general to a method and system for data processing and, in particular, to a method and system for emulating differing architectures in a data processing system. Still more particularly, the present invention relates to a method and system for address translation during emulation of guest instructions in a data processing system.

2. Description of the Related Art

The PowerPC™ architecture is a high-performance reduced instruction set (RISC) processor architecture that provides a definition of the instruction set, registers, addressing modes, and the like, for a family of computer systems. The PowerPC™ architecture is somewhat independent of the particular construction of the microprocessor chips or chips utilized to implement an instance of the architecture and has accordingly been constructed in various implementations, including the PowerPC 601™, 602™, 603™, and 604™. The design and operation of these processors have been described in published manuals such as the *PowerPC 604™ RISC Microprocessor User's Manual*, which is available from IBM Microelectronics as Order No. MPR604UMU-01 and is incorporated herein by reference.

As is true for many contemporary processors, a RISC architecture was chosen for the PowerPC™ because of the inherently higher performance potential of RISC architectures compared to CISC (complex instruction set computer) architectures. While it is desirable to optimize the design of a RISC processor to maximize the performance of the processor when executing native RISC instructions, it is also desirable to promote compatibility by accommodating commercial software written for CISC processors such as the Intel x86 and Motorola 68K.

Accordingly, an emulator mechanism can be incorporated into a PowerPC™ processor as disclosed in above-referenced Ser. No. 08/591,291 now U.S. Pat. No. 5,732,235 and Ser. No. 08/581,793 now U.S. Pat. No. 5,758,140. The disclosed emulation mechanism allows guest instructions (e.g., variable-length CISC instructions) to be emulated by executing corresponding semantic routines formed from native RISC instructions. Thus, the processor is required to manage two distinct instruction streams: a guest instruction stream containing the instructions to be emulated and a native instruction stream containing the native instructions within the semantic routines utilized to emulate the guest instructions. In order to maintain high performance when emulating guest instructions, an efficient mechanism is needed within the processor for managing both the guest and native instruction steams, with provision for branching, address translation buffer management, and exception handling.

The architecture of the Intel x86 line of microprocessors is described in *Microprocessors. Vol. I and Vol. II*, 1993, published by Intel Corporation as Publ. No. 230843. The Intel x86 instruction set is characterized in that the instructions are of variable length, from one byte in length to several bytes, and that arithmetic and logic operations can include a memory access (i.e., the operations can be memory-to-memory operations). In addition, complex addressing modes such as memory indirect are allowed. The architecture of the Motorola 68K line of microprocessors, which is described in various published documents such as *MC68030—Enhanced 32-bit Microprocessor User's Manual*, Prentice Hall, 1990, similarly uses complex addressing modes and variable-length instructions that can specify memory-to-memory operations.

The differences between RISC and CISC instruction sets also results in the utilization of diverse memory management methods and structures. For example, the Intel x86 architecture implements memory segmentation and paging in a manner that permits variable-length segments, while the PowerPC™ architecture employs fixed-length memory segments. Due to this and many other differences between the segmentation and paging mechanisms of the PowerPC and x86 architectures, the contents of page table and translation buffer entries are calculated using quite different logic.

Accordingly, the present invention includes the recognition that it would be desirable to provide a method and apparatus that permit a native (e.g., PowerPC™) architecture to use page table and translation buffer entries that are tailored to the memory management scheme of the guest (e.g., x86) instructions while performing the actual access to physical memory utilizing the native addressing mechanism.

SUMMARY OF THE INVENTION

It is therefore one object of the present invention to provide an improved method and system for data processing.

It is another object of the present invention to provide a method and system for a method and system for emulating differing architectures in a data processing system.

It is yet another object of the present invention to provide a method and system for address translation during emulation of guest instructions in a data processing system.

The foregoing objects are achieved as is now described. According to one embodiment, an emulation mechanism for a host computer system allows guest instructions to be

executed by semantic routines made up of native instructions. The native instructions for the host processor are of a particular format, such as that specified by a RISC architecture, whereas the guest instructions are in a format for a different computer architecture, such as variable-length CISC instructions. The processor includes an emulator unit for fetching and processing the guest instructions that utilizes a multiple-entry queue to store the guest instructions currently fetched in order of receipt. Each entry in the queue includes an offset that indicates the location in memory of the semantic routine for the associated guest instruction, immediate data (if any) for the guest instruction, the length of the corresponding semantic routine, a condition field indicating results of arithmetic/logic operations by a guest instruction, valid bits, and other pertinent data. The processor executes a semantic routine in response to the entries in the queue, using the content of the entry to fetch the semantic routine. An entry is removed from the queue when the semantic routine for the associated guest instruction has been completed by the processor.

The memory management scheme for the guest instructions is different from that of the native instructions; accordingly, the translation of guest virtual addresses to guest real addresses is based on a different logic scheme. According to the present invention, a guest logical address is translated into a guest real address, which is thereafter translated into a native physical address. A semantic routine that emulates a guest instruction that accesses memory can then be executed utilizing the native physical address.

The above as well as additional objects, features, and advantages of the present invention will become apparent in the following detailed written description.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself however, as well as a preferred mode of use, further objects and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

FIG. 1 depicts an illustrative embodiment of a data processing system with which the method and system of the present invention may advantageously be utilized;

FIG. 2 illustrates a more detailed block diagram of the processor depicted in FIG. 1;

FIG. 3 depicts a more detailed block diagram of the emulation assist unit (EAU) in the processor of FIG. 2;

FIG. 4 illustrates a more detailed block diagram of the guest instruction queue within the EAU depicted in FIG. 3;

FIG. 5 is a diagram of a memory management scheme for CPU 4 utilized in the illustrative embodiment of FIG. 2;

FIG. 6 is a diagram of a memory management scheme for the guest instructions used in the embodiment of FIG. 1–3;

FIG. 7 is a diagram of an address translation scheme for guest instructions in the embodiment of FIGS. 1–6;

FIG. 8 is a high level logical flowchart of a method for generating a guest TLB entry in accordance with the illustrative embodiment; and

FIG. 9 is a diagram of certain registers used in the address translation scheme illustrated in FIGS. 7 and 8.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENT

With reference now to the figures and in particular with reference to FIG. 1, there is depicted a high level block diagram of a data processing system 2 in accordance with the present invention. As illustrated, data processing system 2, which can comprise a desktop computer system, includes one or more CPUs 4, which are connected to the other components of data processing system 2 in a standard hardware configuration. For example, CPUs 4 can be interconnected to main memory 6 via a memory controller 5 and system bus 7. System bus 7 is also coupled to PCI (Peripheral Component Interconnect) bus 12 by a PCI host bridge 11, which permits communication between the devices coupled to system bus 7 and peripherals 10 and I/O components 8. Although for the purpose of illustration, the present invention is described below with reference to an illustrative embodiment in which CPU 4 is implemented with one of the PowerPC™ line of processors manufactured by International Business Machines Corporation, it should be understood that a variety of other processors could alternatively be employed.

When implemented as a PowerPC™ processor, each CPU 4 preferably comprises a single integrated circuit superscalar microprocessor, including various registers, buffers, execution units, and functional units that operate according to reduced instruction set computing (RISC) techniques. Each CPU 4 executes RISC instructions within the PowerPC™ instruction set architecture (e.g., the instructions forming application program 16 and operating system/kernel 18) from a memory map 14. The PowerPC™ instruction set architecture native to CPU 4 is defined in a number of publications such as *PowerPC™ User Instruction Set Architecture* and *PowerPC™ 603 RISC Microprocessor User's Manual* (Order No. MPR603UMU-01), both available from IBM Microelectronics. RISC instructions, such as those defined by the PowerPC™ instruction set architecture, can be characterized as having a fixed instruction length (e.g., 32-bits), including only register-to-register and register-to-memory operations and not memory-to-memory operations, and being executed without microcoding, often in one machine cycle.

Each CPU 4 is further adapted in accordance with the present invention to execute guest instructions (e.g., CISC instructions or some other instruction set that is not native to CPU 4) by emulation. As described further hereinbelow, guest instructions 20 are each emulated by fetching and executing one or more semantic routines 19, which each contain two or more native instructions. For example, a guest instruction 20 might be a memory-to-memory CISC instruction such as:

```
ADD MEM1, MEM2, MEM3
```

meaning "add the contents of memory location #1 to the contents of memory location #2 and store the result in memory location #3." A semantic routine 19 to emulate this guest CISC instruction might contain the following native RISC instructions:

```
LOAD REG1, MEM1
LOAD REG2, MEM2
ADD REG3, REG2, REG1
STORE REG3, MEM3
```

This exemplary semantic routine 19 loads the contents of memory locations #1 and #2 into registers #1 and #2, respectively, adds the contents of registers #1 and #2, stores the result of the addition in register #3, and stores the contents of register #3 to memory location #3. As further

illustrated in FIG. 1, memory map 14 preferably further includes semantic routine synthesis code 22, which comprises user-level code that can be utilized to synthesize a semantic routine corresponding to a guest instruction if such a semantic routine is not already one of the semantic routines in area 19.

Referring now to FIG. 2, there is illustrated a more detailed block diagram of CPU 4. As depicted, CPU 4 is coupled to system bus 12 via a bus interface unit (BIU) 30 that controls the transfer of information between CPU 4 and other devices that are coupled to system bus 12. BIU 30 is also connected to instruction cache 32 and data cache 34. Both instruction cache 32 and data cache 34 are high-speed caches which enable CPU 4 to achieve a relatively fast access time to instructions and data previously transferred from main memory 6, thus improving the speed of operation of data processing system 2. Instruction cache 32 is further coupled to sequential fetcher 36, which fetches native instructions from instruction cache 32 during each execution cycle. Sequential fetcher 36 transmits branch instructions fetched from instruction cache 32 to branch processing unit (BPU) 38 for execution, but temporarily buffers sequential instructions within instruction queue 40. The sequential instructions stored within instruction queue 40 are subsequently dispatched by dispatch unit 42 to the sequential execution circuitry of CPU 4.

In the depicted illustrative embodiment, the sequential execution circuitry of CPU 4 includes three (or more) execution units, namely, fixed-point unit (FXU) 50, load/store unit (LSU) 52, and floating-point unit (FPU) 54. Each of these three execution units can execute one or more classes of native instructions, and all execution units can operate concurrently during each processor cycle. For example, FXU 50 performs fixed-point mathematical operations such as addition, subtraction, ANDing, ORing, and XORing, utilizing source operands received from specified general purpose registers (GPRs) 60 or GPR rename buffers 62. Following the execution of a fixed-point instruction, FXU 50 outputs the data results of the instruction to GPR rename buffers 62, which provide temporary storage for the data results until the data results are written to at least one of the GPRs 60 during the writeback stage of instruction processing. Similarly, FPU 54 performs floating-point operations, such as floating-point multiplication and division, on source operands received from floating-point registers (FPRs) 64 or FPR rename buffers 66. FPU 54 outputs data resulting from the execution of floating-point instructions to selected FPR rename buffers 66, which temporarily store the data results until the data results are written to selected FPRs 64 during the writeback stage of instruction processing. As its name implies, LSU 52 executes floating-point and fixed-point instructions which either load data from memory (i.e., either data cache 34 or main memory 6) into selected GPRs 60 or FPRs 64 or which store data from a selected one of GPRs 60, GPR rename buffers 62, FPRs 64, or FPR rename buffers 66 to data cache 34 or main memory 6.

CPU 4 employs both pipelining and out-of-order execution of instructions to further improve the performance of its superscalar architecture. Accordingly, multiple instructions can be simultaneously executed by BPU 38, FXU 50, LSU 52, and FPU 54 in any order as long as data dependencies and antidependencies are observed between sequential instructions. In addition, instructions are processed by each of FXU 50, LSU 52, and FPU 54 at a sequence of pipeline stages, including fetch, decode/dispatch, execute, finish and completion/writeback. Those skilled in the art should appreciate, however, that some pipeline stages can be reduced or combined in certain design implementations.

During the fetch stage, sequential fetcher 36 retrieves one or more native instructions associated with one or more memory addresses from instruction cache 32. As noted above, sequential instructions fetched from instruction cache 32 are stored by sequential fetcher 36 within instruction queue 40. In contrast, sequential fetcher 36 removes (folds out) branch instructions from the instruction stream and forwards them to BPU 38 for execution. BPU 38 preferably includes a branch prediction mechanism, which in an illustrative embodiment comprises a dynamic prediction mechanism such as a branch history table, that enables BPU 38 to speculatively execute unresolved conditional branch instructions by predicting whether or not the branch will be taken.

During the decode/dispatch stage, dispatch unit 42 decodes and dispatches one or more native instructions from instruction queue 40 to an appropriate one of sequential execution unit 50, 52, and 54 as dispatch-dependent execution resources become available. These execution resources, which are allocated by dispatch unit 42, include a rename buffer within GPR rename buffers 60 or FPR rename buffers 66 for the data result of each dispatched instruction and an entry in the completion buffer of completion unit 44.

During the execute stage, execution units 50, 52, and 54 execute native instructions received from dispatch unit 42 opportunistically as operands and execution resources for the indicated operations become available. In order to minimize dispatch stalls, each one of the execution units 50, 52, and 54 is preferably equipped with a reservation table that stores dispatched instructions for which operands or execution resources are unavailable.

After the operation indicated by a native instruction has been performed, the data results of the operation are stored by execution units 50, 52, and 54 within either GPR rename buffers 62 or FPR rename buffers 66, depending upon the instruction type. Then, execution units 50, 52, and 54 signal completion unit 44 that the execution unit has finished an instruction. In response to receipt of a finish signal, completion unit 44 marks the completion buffer entry of the instruction specified by the finish signal as complete. Instructions marked as complete thereafter enter the writeback stage, in which instructions results are written to the architected state by transferring the data results from GPR rename buffers 62 to GPRs 60 or FPR rename buffers 66 to FPRs 64, respectively. In order to support precise exception handling, native instructions are written back in program order.

As illustrated in FIG. 2, in order to facilitate the emulation of guest instructions, CPU 4 includes emulation assist unit (EAU) 70, which is shown in greater detail in FIG. 3. As illustrated in FIG. 3, EAU 70 includes a number of special purpose registers (SPRs) 76 for storing, among other things, the logical base address of segments of guest address space containing guest instructions. SPRs 76 include a code segment base register (CSBR) 75 that stores the base address of the current segment and an offset to the current guest instruction. EAU 70 further includes an instruction prefetch unit 74 for fetching guest instructions from data cache 34 and an instruction buffer 72 for temporarily storing guest instructions retrieved from data cache 34. In addition, EAU 70 includes an instruction decoder 78 for decoding guest instructions, a guest branch unit 80 for executing guest branch instructions, tag generation unit 90, which generates opcode tags for each sequential guest instruction, guest cache 92, which stores a semantic routine (SR) offset in association with each of a plurality of opcode tags, a guest

instruction queue 100 for storing information associated with guest instructions, and a guest dispatch unit 94 that provides SR addresses to sequential fetcher 36.

Referring now to FIG. 4, there is illustrated a more detailed view of guest instruction queue 100, which provides a synchronization point between the guest instruction stream and native instruction stream. As will become apparent from the following description, the provision of guest instruction queue 100 permits guest instructions emulated by CPU 4 to be pre-processed so that the latency associated with the various emulation pipeline stages can be overlapped.

In the illustrative embodiment, guest instruction queue 100 contains five entries 110, which each include the following fields 112–134:

- V_I: indicates whether the content of immediate field 116 is valid
- V_O: indicates whether the content of SR offset field 118 is valid
- Immediate: stores immediate data that is specified by the guest instruction and is passed as a parameter to the corresponding semantic routine
- SR offset: offset between the base address of the guest instruction (which is maintained in CSBR 75) and the corresponding semantic routine
- L: length of semantic routine in native instructions
- GIP: offset pointer from CSBR 75 to guest instruction in guest address space
- DE: indicates whether two guest instruction queue entries (and two semantic routines) are utilized in the emulation of a single guest instruction
- S: indicates whether the guest instruction is in a speculative (i.e., predicted) execution path in the guest instruction stream
- BBID: unique basic block ID number sequentially assigned to each semantic routine from pool of BBIDs
- GCR: guest condition register that indicates conditions (e.g., equal/not equal) that may be utilized to predict subsequent guest branch instructions
- CRM: guest condition register mask that indicates which bits in the GCR field will be altered by the guest instruction
- V_B: indicates whether the semantic routine native instruction that will set the value of GCR field 130 has executed

As depicted in FIG. 4, guest instruction queue 100 has an associated emulation instruction pointer register (EIPR) 140, preferably implemented as a software-accessible special purpose register (SPR), which contains the offset from the base address specified by CSBR 75 to the current guest instruction that is being interpreted. EAU 70 updates the contents of EIPR 140 in response to the execution of a newly-defined "guest dispatch completion" (gdispc) instruction in the native instruction set and in response to the execution of a guest branch instruction by guest branch unit 80 without invoking a semantic routine. Another special purpose register, emulation instruction pointer next (EIP_NEXT) register 142, contains the offset from the base address specified in CSBR 75 to the next guest instruction that will be interpreted. EAU 70 updates the contents of EIP_NEXT register 142 when a gdispc instruction is executed, when a special move to SPR instruction (i.e., mtspr[EIP_NEXT]) is executed having EIP_NEXT register 142 as a target, and when a guest branch or guest NOOP instruction is emulated without invoking a semantic routine. These two offset pointers permit the state of the guest instruction stream to be easily restored following a context switch, for example, when returning from an exception. That is, by saving both the current EIP and the next EIP, the guest instruction under emulation at the time of the interrupt, which is pointed to by the current EIP, does not need to be reexecuted to compute the next EIP if both the current EIP and next EIP are saved.

Guest instruction queue 100 also has an associated predicted taken flag (PTF) 144, which indicates whether an unresolved guest branch instruction was predicted as taken and therefore whether sequential guest instructions marked as speculative (i.e., S field 126 is set) are within the target or sequential execution path.

Xflags 150 is an architected condition register for which GCR 130 in each of entries 110 is a "renamed" version. When an entry 110 is removed from the bottom of guest instruction queue 100, the bits within Xflags 150 specified by CRM 132 in that entry 110 are updated by the corresponding bit values in GCR 130. Xflags 150, GCR fields 130, CRM fields 132, and V_B fields 134 (and the associated access circuitry), which are identified in FIG. 3 simply as guest flags 83, can be referenced by guest branch unit 80 to resolve guest branch instructions as described further herein below.

In cases in which each guest instruction is emulated by executing a single semantic routine, each guest instruction is allocated only a single entry 110 within guest instruction queue 100. However, in some circumstances more than one entry 110 may be allocated to a single sequential guest instruction. For example, in an embodiment in which the guest instructions are x86 instructions, many sequential guest instruction comprise two distinct portions: a first portion that specifies how the addresses of the source(s) and destination of the data are determined and a second portion that specifies the operation to be performed on the data. In such cases, a first semantic routine is utilized to emulate the portion of instruction execution related to the determination of the data source and destination addresses and a second semantic routine is utilized to emulate the portion of instruction execution related to performing an operation on the data. Accordingly, the guest instruction is allocated two entries 110 in guest instruction queue 100—a first entry containing information relevant to the first semantic routine and a second entry containing information relevant to the second semantic routine. Such dual entry guest instructions are indicated within guest instruction queue 100 by setting DE (dual entry) field 124 in the older (first) of the two entries 110. Setting the DE field ensures that both entries 110 will be retired from guest instruction queue 100 when both semantic routines have completed (i.e., in response to a gdispc instruction terminating the second semantic routine). The emulation of guest instructions utilizing two semantic routines advantageously permits some semantic routines to be shared by multiple guest instructions, thereby reducing the overall memory footprint of semantic routines 19.

The ordering of the entries 110 in guest instruction queue 100 is maintained by current entry pointer 95, which points to the oldest entry in guest instruction queue 100, and next entry pointer 97, which points to the next oldest entry. In response to a fetch or completion of a gdispc instruction, the guest instruction queue entry indicated by current entry pointer 95 is retired and both current entry pointer 95 and next entry pointer 97 are updated. Thus, entries are consumed from the "bottom" and inserted at the "top" of guest instruction queue 100.

With reference now to FIGS. 2–4, the operation of EAU 70 will now be described.

EAU INITIALIZATION

To initialize EAU 70 for emulation, the address offset to the first guest instruction to be emulated is loaded into EIP_NEXT register 142 by executing a native move to SPR (mtspr) instruction having EIP_NEXT register 142 as a target (i.e., mtspr[EIP_NEXT] in the PowerPC™ instruction set). In a preferred embodiment, this native instruction is equivalent to a guest branch always instruction since the function of such a guest branch instruction would be to load EIP_NEXT register 142 with a pointer to the next guest instruction to be executed (i.e., the offset value within CSBR 75). V_I field 112 and V_O field 114 of the oldest entry 110 in guest instruction queue 100 are both cleared in response to the mtspr[EIP_NEXT] instruction. Thereafter, prefetching of guest instruction from data cache 34 can be triggered utilizing a gdispc instruction.

As an aside, V_I field 112 and V_O field 114 of the oldest entry 110 in guest instruction queue 100 are also cleared in response to mtspr[EIP] and mtspr[CSBR] instructions, as well as when a guest branch instruction is resolved as mispredicted.

GUEST INSTRUCTION PREFETCHING

As noted above, prefetching of guest instructions from data cache 34 is triggered by placing a gdispc instruction in the native instruction stream. When fetched by sequential fetcher 36, the gdispc instruction acts as an interlock that stalls fetching by sequential fetcher 36 until V_O field 114 of the oldest entry 110 in guest instruction queue 100 is set. In response to the stall of sequential fetcher 36, instruction prefetch unit 74 in EAU 70 makes a fetch request to data cache 34 for the guest instruction at the address specified by the base address and offset contained in CSBR 75.

GUEST INSTRUCTION DECODING

Guest instructions supplied by data cache 34 in response to fetch requests from instruction prefetch unit 74 are temporarily stored in instruction buffer 72 and then loaded one at a time into instruction decoder 78, which at least partially decodes each guest instruction to determine the instruction length, whether the guest instruction is a branch instruction, and the immediate data of the guest instruction, if any.

GUEST BRANCH INSTRUCTION PROCESSING

If instruction decoder 78 determines that a guest instruction is a branch instruction, the guest branch instruction is forwarded to guest branch unit 80 for processing after allocating the guest branch instruction the oldest unused entry 110 of guest instruction queue 100. (In an alternative embodiment, guest instruction ordering can be maintained without assigning guest instruction queue entries to guest branch instructions). Guest branch unit 80 first attempts to resolve a conditional guest branch instruction with reference to guest flags 83. If the bit(s) upon which the guest branch depends are set within CRM field 132 and V_B field 134 is marked valid in the entry 110 corresponding to the immediately preceding sequential guest instruction, guest branch unit 80 resolves the guest branch instruction by reference to GCR field 130 of the entry 110 of immediately preceding sequential guest instruction. If, however, the relevant bit(s) within CRM 132 in the entry 110 corresponding to the immediately preceding sequential guest instruction are not set, the guest branch instruction is resolved by reference to the newest preceding entry 110, if any, having the relevant bits set in CRM field 132 and V_B field 134 marked as valid, or failing that, by reference to Xflags 150. On the other hand, guest branch unit 80 predicts (i.e., speculatively executes) a conditional guest branch instruction by reference to conventional branch history table (BHT) 82 if V_B field 134 is marked invalid in the newest preceding entry 110 in which the bit(s) relevant to the guest branch are set in GCR field 130. The guest instruction at the address of the resolved or predicted execution path is thereafter fetched from data cache 34 via instruction prefetch unit 74. Further details about the processing of guest branch instructions are found in Ser. No. 08/934,644 now U.S. Pat. No. 5,870,575 and Ser. No. 08/934,857, which were referenced hereinabove.

SEQUENTIAL GUEST INSTRUCTION PROCESSING

If the guest instruction decoded by instruction decoder 78 is a sequential instruction, at least the oldest unused entry 110 of guest instruction queue 100 is allocated to the guest instruction. As illustrated in FIG. 3, instruction decoder 78 then stores the immediate data, if any, and the offset pointer to the guest instruction into immediate field 116 and GIP field 122, respectively, of the allocated entry 110. In response to instruction decoder 78 loading immediate data into immediate field 116, V_I field 112 is set.

The sequential guest instruction is then forwarded from instruction decoder 78 to tag generation unit 90, which converts the guest instruction into a unique opcode tag. According to a preferred embodiment, different opcode tags are utilized not only to distinguish between different guest instructions, but also to distinguish between identical guest instructions that access different registers. Thus, different opcode tags are utilized for guest divide (gdiv) and guest multiply (gmult) instructions, as well for gmult R3,R2,R1 and gmult R4,R2,R1 instructions, which target different registers. The unique opcode tag produced by tag generation unit 90 forms an index into guest cache 92 that selects a particular cache entry containing an offset utilized to determine the effective address of the semantic routine corresponding to the guest instruction.

As indicated, in the illustrative embodiment, guest cache 92 comprises a four-way set associative cache having 256 lines that each contain four 4 Kbyte entries. A miss in guest cache 92 generates a user-level interrupt, which is serviced by executing semantic routine synthesis code 22. As described above, semantic routine synthesis code 22 synthesizes a semantic routine corresponding to the guest instruction from native instructions and stores the semantic routine in area 19 of memory map 14. The offset from the base address of the guest instruction to the location of the newly synthesized semantic routine is then stored in guest cache 92 for subsequent recall. Because guest instruction sets are typically fairly stable, it is typical for guest cache 92 to achieve hit rates above 99%.

In response to the semantic routine (SR) offset being located (or stored) in guest cache 92, the SR offset is stored in SR offset field 118 of the allocated entry 110, thereby causing V_O field 114 to be marked as valid. By the time V_O is set to signify that the content of SR offset field 118 is valid, L field 120, DE field 124, S field 126, BBID field 128, and CRM field 132 are also valid within the allocated entry 110. As noted above, GCR field 130 is indicated as valid separately by V_B field 134.

When V_O field 114 of the oldest entry 110 in guest instruction queue 100 is set by the processing of the first guest instruction in EAU 70 at emulation startup, the value in EIP_NEXT register 142 is transferred to EIPR 140, signifying that the oldest (i.e., first) instruction in guest instruction queue 100 is the guest instruction currently being processed. In response to this event, guest dispatch unit 94 transmits the SR offset in SR offset field 118 to sequential fetcher 36, which begins to fetch native instructions within the semantic routine corresponding to the first guest instruction. As illustrated in FIG. 4, EAU 70 tracks the guest

instruction for which the semantic routine is being fetched utilizing fetch PTR 152 in guest dispatch unit 94.

SEMANTIC ROUTINE PROCESSING

Semantic routine (i.e., native) instructions that are within the standard instruction set of CPU 4 are processed by CPU 4 as described above with reference to FIG. 2. Special instructions inserted into the native instruction set to support guest instruction emulation are handled as described below.

In order to connect guest instructions into a continuous guest instruction stream, a gdispc instruction is preferably inserted at the end of each semantic routine, if the guest instructions are each represented by a single semantic routine, or at the end of the last semantic routine corresponding to the guest instruction, if the guest instruction is emulated by multiple semantic routines. The gdispc instruction is preferably defined as a special form of a native branch instruction so that when fetched from instruction cache 32 by sequential fetcher 36 a gdispc instruction is folded out of the native instruction stream and passed to BPU 38. In response to detecting the gdispc instruction, BPU 38 asserts signal line 96. Guest dispatch unit 94 responds to the assertion of signal line 96 by removing all of the entries 110 corresponding to the current guest instruction from guest instruction queue 100 and by passing the semantic routine offset stored within the next entry to sequential fetcher 36 via signal lines 98. As described above, sequential fetcher 36 then computes the effective address (EA) of the semantic routine corresponding to the next guest instruction by adding the semantic routine offset to the guest instruction's base address and fetches the semantic routine from memory for execution by CPU 4.

When multiple semantic routines are utilized to emulate a single guest instruction, semantic routines other than the final semantic routine are terminated by a "guest dispatch prolog completion" (gdispp) instruction, which is a variant of the gdispc instruction. In general, the gdispp instruction is processed like the gdispc instruction. For example, like the gdispc instruction, the gdispp instruction triggers the fetching of the next semantic routine. In addition, V_O field 114 within the guest instruction queue entry 110 corresponding to the semantic routine containing a gdispp instruction must be set in order for the gdispp instruction to be executed. However, in contrast to the processing of a gdispc instruction, the completion of a gdispp instruction does not trigger the removal of an entry 110 from guest instruction queue 100 or the updating of EIPR 140 and EIP_NEXT register 142.

Another special instruction inserted into the native instruction set as a form of add instruction is the guest add immediate prolog [word or half word] (gaddpi[w,h]) instruction. The function of the gaddpi[w,h] instruction is to add the immediate data specified in the first of two guest instruction queue entries allocated to a guest instruction with the value in a specified GPR 60 and store the sum in another GPR 60. Accordingly, V_I field 112 for the first entry 110 must be set in order to permit the corresponding semantic routine to execute.

A similar guest add immediate completion [word or half word] (gaddci[w,h]) instruction is utilized to add the immediate data stored in the second of two guest instruction queue entries allocated to a guest instruction with value of a specified GPR 60 and store the sum in another GPR 60. V_I field 112 for the second entry 110 must be set in order for the corresponding semantic routine to execute.

INTERRUPT AND EXCEPTION HANDLING

In response to either a guest instruction or native instruction exception, a non-architected exception flag is set that disables guest instruction fetching by instruction prefetch unit 74. At a minimum, the context of the guest instruction stream is saved during interrupt/exception handling and restored upon returning from the interrupt/exception by saving the contents of EIPR 140 and EIP_NEXT register 142 in SPRs 76. As a practical matter, it is preferable to save the entire bottom entry 110 of guest instruction queue 100 in SPRs 76 in order to expedite the restart of emulation following the interrupt/exception.

Prefetching of guest instructions from data cache 34 following a return from interrupt can be triggered by the execution of either a gaddpi[w,h] instruction or gaddci[w,h] instruction, which interlocks with and stalls sequential fetcher 36 until V_I field 112 of the appropriate entry 110 in guest instruction queue 100 is set. Guest instruction prefetching may also be restarted through the execution of a gdispc instruction or gdispp instruction. The execution of a gdisp[p,c] or gadd[p,c]i[w,h] instruction clears the exception flag.

MEMORY MANAGEMENT

Referring again to FIG. 2, instruction cache 32 (and main memory 6) is accessed via an instruction memory management unit (IMMU) 58, and likewise data cache 34 (and main memory 6) is accessed via a data memory management unit (DMMU) 56. Each of memory management units 56 and 58 has its own respective translation lookaside buffer, so there is an ITLB 59 and a separate DTLB 57. TLBs 57 and 59 each contain copies of page table entries from the page tables in memory 6, which correlate real (physical) addresses of pages in memory with logical (effective) addresses generated by CPU 4 for instructions and data. Since the memory management scheme for the guest (e.g., Intel x86) instruction architecture may differ significantly from the native PowerPC™ scheme, memory management units (MMUs) 56 and 58 include address translation facilities for guest instructions that reference memory.

Referring now to FIG. 5, there is depicted a diagram of the scheme utilized by MMUs 56 and 58 to translate logical (effective) addresses into physical addresses while processing native instructions. As illustrated, in response to receipt of a 32-bit logical (effective) address 60 by one of MMUs 56 and 58, a determination is made whether address translation is enabled. If not, the logical address is utilized as the physical address, as indicated at reference numeral 174. If, however, address translation is enabled, the MMU utilizes 4 high order bits from the logical address to select a segment register that contains a 24-bit segment address, as depicted at reference numeral 162. In parallel with the selection of the segment register, the logical address is compared with the address ranges defined in a Block Address Translation (BAT) array as illustrated at reference numeral 164. If the logical address falls with an address range defined in the BAT array, then block address translation is performed to obtain a 32-bit physical address as indicated at reference numeral 166.

However, if no match is found for the logical address in the BAT array, a control bit in the descriptor of the selected segment is tested to determine if the access is to memory or to I/O controller interface space. If the bit in the segment descriptor indicates that the access is to I/O controller interface space, I/O controller interface translation is performed to obtain a 32-bit I/O controller interface address, as shown at reference numeral 168. Otherwise, page address translation is performed by concatenating the 24-bit segment address with the low order 28 bits of the logical address to produce a 52-bit virtual address 170, and by thereafter translating this 52-bit virtual address into a 32-bit physical

address, if possible, by reference to a page table entry (as indicated at reference numeral 172). If the required page table entry (PTE) is present in the relevant one of DTLB 57 and ITLB 59, the physical address corresponding to the logical address is immediately available. However, if a TLB miss occurs, an exception is taken, and the page table in memory is searched for the matching PTE.

Regardless of whether page address translation, block address translation, or direct address translation is performed, the resulting 32-bit physical address can then be utilized to access one of instruction cache 32 and data cache 34, and if a cache miss occurs, main memory 6.

With reference now to FIG. 6, the conventional two-level address translation scheme of the Intel x86 architecture is illustrated in diagram form. As depicted, a 32-bit linear address 180 is translated into a physical address by first partitioning linear address 180 into a 10-bit directory field 182, a 10-bit table field 184, and a 12-bit offset field 186. The value of directory field 182 is utilized as an offset that, when added to a root address stored in control register 192, accesses an entry 194 in page directory 190. Page directory entry 194 contains a pointer that identifies the base address of page table 200. The value of table field 184 forms an offset pointer that, when added to the value of directory entry 194, selects a page table entry 202 that specifies the base address of a page 204 in memory. The value of offset field 186 then specifies a particular physical address 210 within page 204.

As depicted in FIG. 6, the 20 high order bits of linear address 180 are also utilized in parallel to search for a matching page table entry in x86 TLB 220. If a match is found in x86 TLB 220, the matching page table entry is utilized to perform linear-to-real address translation in lieu of page directory 190 and page table 200, which require memory accesses.

By comparison of FIGS. 5 and 6, it should be apparent that guest instructions require a different page address translation scheme than native instructions. Accordingly, referring now to FIG. 7, a high level diagram of the method employed by the present invention to translate guest linear addresses into native real (physical) addresses is shown. According to the present invention, each guest linear address 240, which may be the target address of a guest instruction fetch, guest data load, or guest data store, is first translated into a guest real address 242 using the logical equivalent of the address translation scheme depicted in FIG. 6. A simple mapping translation, such as adding a fixed offset (which may be zero), is then performed to obtain native effective address 244. Subsequently, a native physical (real) address 246 is calculated using the logical equivalent of the process illustrated in FIG. 5.

With reference now to FIG. 8, a high level logical flowchart is provided that illustrates how the process of guest address translation shown in FIG. 7 is preferably implemented within CPU 4. In the preferred embodiment, all guest storage accesses (i.e., guest instruction fetches, data loads, and data stores) are treated as data accesses by CPU 4, and are accordingly accessed using DMMU 56. The registers utilized by DMMU 56 to translate guest addresses are illustrated in FIG. 9.

As depicted in FIG. 8, the process of guest address translation begins at block 250 in response to receipt of a guest linear address by DMMU 56. DMMU 56 distinguishes between guest linear addresses and native effective addresses received as inputs by the presence (or absence) of a bit generated when the memory access instruction was decoded. The process proceeds from block 250 to block 252, which illustrates a determination of whether or not DTLB 57 contains a "guest" entry that maps the guest linear address to a native physical address. In a preferred embodiment, DTLB 57 contains both native entries as well as guest entries, which are marked, for example, with an "X" bit set to one. However, in other embodiments, DTLB 57 may include separate TLBs for native and guest entries. In either case, the determination depicted at block 252 may be made, for example, by comparing the 32-bit guest linear address with the first 32 bits of each guest entry. In response to a hit in DTLB 57, the process proceeds from block 252 to block 253, which illustrates DMMU 56 utilizing the native real page address (described below) in the matching DTLB entry to calculate a native physical address. Next, as shown at block 254, DMMU 56 utilizes the native physical address to access the requested data within either data cache 34 or main memory 6. Thereafter, the process terminates at block 270.

Referring again to block 252, in response to a miss in DTLB 57, a guest DTLB miss exception is generated. Before branching to appropriate exception handler, hardware within CPU 4 saves the address of the semantic routine native instruction at which the DTLB miss exception occurred in one of SPRs 76 designated as EAU_SRR0 296. In a preferred embodiment, this native instruction address is contained in a completion buffer entry within completion unit 44. In addition, hardware saves the 20 high order bits of the guest linear address 180 that caused the DTLB miss in one of SPRs 76 designated as guest page index (GPI) register 282. Execution by CPU 4 then branches to a user-level exception handler located at a predetermined offset from the value stored in guest branch register (GBR) 298. In order to simplify the exception handling logic, the offset of the guest DTLB miss exception handler is preferably equal to the offset assigned to the native DTLB miss exception handler.

As illustrated at block 260 of FIG. 8, the guest DTLB exception handler utilizes the logical equivalent of the address translation process depicted in FIG. 6 to generate a guest real address 242 from the guest linear address stored in GPI register 282. The guest real address 242 is then stored within a general purpose register GPRy. As indicated at block 262, a fixed offset value (which may be zero) is added to the guest real address to produce a native effective address, which is then stored in one of GPRs 60 designated as GPRx 290.

The process then proceeds from block 262 to block 264, which illustrates the execution of a "guest TLB load" (gtlbld) instruction in the exception handler routine. The gtlbld instruction is a newly defined operation in the native instruction set that builds the first word of a guest TLB entry. As illustrated in FIG. 9, the execution of the gtlbld instruction causes hardware within CPU 4 to load the conventional PPC-DCMP register 286 with bits 4–9 (the abbreviated page index (API)) of GPRx 290 and with the value stored in the native segment register 288 selected by the four most significant bits of GPRx 290. Thus, PPC-DCMP register 286 has the same format as the first word of a conventional native DTLB entry. The execution of the gtlbld instruction also causes the hardware of CPU 4 to load selected fields of (guest) X-DCMP register 280 with the 4 high order bits of GPI register 282 and with the 19-bit value of gSegID 284, which, similar to the virtual page numbers stored in native segment registers 288, specifies high order bits of a virtual page number assigned to guest instructions and data. The gtlbld instruction also loads X-DMISS register 292 with the content of GPI register 282, which stores the 20 high order bits of the guest linear address 180 that caused the DTLB

miss. Finally, as illustrated at block 266 of FIG. 8, execution of the gtlbld instruction generates a hardware exception that invokes the conventional PowerPC™ supervisor-level page table walk routine. The page table walk routine determines the PowerPC™ (native) real page address to which the guest linear address maps and creates a guest entry in DTLB 57 to translate the guest linear address.

The table walk routine begins by comparing the value of PPC-DCMP register 286 against the first word of page table entries in memory (data cache 34 and main memory 6) until a match is found. In response to finding a matching entry in the page table, the supervisor-level table walk routine loads real page address (RPA) register 294 with the second word of the matching page table entry. A native TLB load (tlbld) instruction in the supervisor-level table walk routine is then executed. In response to the tlbld instruction, the hardware of CPU 4 selects an entry within DTLB 57, which preferably comprises a 32 entry 2-way set associative TLB array, utilizing bits 15–19 of X-DMISS register 292. In the illustrative embodiment, each DTLB entry comprises a first set including 36 bits and a second set containing 32 bits. As illustrated at block 268 of FIG. 8, the tlbld instruction then causes bits 1–23 of X-DCMP register 280 to be loaded into bits 2–24 of the first set of the selected DTLB entry, bits 4–14 of X-DMISS register 292 to be loaded into bits 25–35 of the first set of the selected DTLB entry, and the value of RPA register 294 to be loaded into the second set of the selected DTLB entry. To signify that the selected DTLB entry contains a guest address translation, the gtlbld instruction also sets an X bit, which in a preferred embodiment comprises bit 1 (i.e., the second most significant bit) of the first set of the selected DTLB entry. The supervisor-level table walk routine then executes a native "return from interrupt" (rfi) instruction and returns control to the user-level exception handler.

Thereafter, the user-level exception handler returns to the emulation of guest instructions utilizing the following sequence of native instructions:

```
mfspr GPRz, EAU_SRRO

mtctr GPRz

bcctr BO[0]=1
```

The mfspr ("move from SPR") instruction loads the effective address of the instruction at which the exception occurred from EAU_SRRO 296 into GPRz. The effective address is then transferred from GPRz into the PowerPC™ count register via the mtctr ("move to count register") instruction. The guest context of EAU 70 is then restored to its pre-exception state by executing the bcctr ("branch conditional to count register") instruction, which causes sequential fetcher 36 to resume fetching native instructions at the specified effective address (i.e., at the point where sequential fetcher 36 discontinued fetching native instructions in response to the DTLB miss exception). When the native instruction at which the exception occurred is again executed, the guest linear address is again passed to DMMU 56 for translation, as illustrated at block 252. This time the guest linear address bits in DTLB 57, and the process proceeds to block 253. Block 253 depicts the formation of the native physical address by concatenating the 19 high order bits of the second set of the matching DTLB entry (i.e., the real page number) with the 12-bit offset of the guest linear address. As depicted at block 254, DMMU 56 then performs the memory access utilizing the native real address. Thereafter, the process terminates at block 270.

While an illustrative embodiment of the present invention has been particularly shown and described, it will be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope hereof. For example, while the present invention has been described with reference to embodiments in which the guest instructions emulated within CPU 4 are x86 CISC instructions, it should be understood that other guest instructions could alternatively be emulated.

What is claimed is:

1. A method of operating a processor which has a native instruction set and emulates instructions in a guest instruction set, said method comprising:

storing, in memory, a series of guest instructions from said guest instruction set, said series including a guest memory access instruction that indicates a guest logical address in guest address space;

for each guest instruction in said series, storing in memory a semantic routine of native instructions from said native instruction set to emulate each guest instruction, said native instructions utilizing native addresses in native address space;

in response to receipt of said guest memory access instruction for emulation, translating said guest logical address into a guest real address and thereafter translating said guest real address into a native physical address; and

executing a semantic routine that emulates said guest memory access instruction utilizing said native physical address.

2. The method of claim 1, wherein said guest memory access instruction comprises one of a guest load instruction and a guest store instruction.

3. The method of claim 1, wherein said guest memory access instruction comprises a guest instruction that initiates fetching of a guest instruction in said series from memory.

4. The method of claim 1, wherein said step of executing a semantic routine that emulates said guest memory access instruction comprises the step of accessing said memory utilizing said native physical address.

5. The method of claim 1, said step of translating said guest real address into a native physical address includes the step of translating said guest real address into a native effective address and then translating said native effective address into said native physical address.

6. The method of claim 1, said processor including a translation lookaside buffer (TLB) containing entries utilized for address translation, wherein said step of translating said guest logical address into a guest real address and thereafter translating said guest read address into a native physical address comprises the steps of:

determining if said translation lookaside buffer includes an entry that can be utilized to obtain said native physical address; and

in response to a determination that said translation lookaside buffer contains an entry that can be utilized to obtain said native physical address, translating said guest logical address into a guest real address and thereafter translating said guest read address into a native physical address utilizing said translation lookaside buffer (TLB) entry.

7. The method of claim 6, said method further comprising the step of:

in response to a determination that said translation lookaside buffer does not contain an entry that can be utilized to obtain said native physical address, creating an entry

that can be utilized to obtain said native physical address in said translation lookaside buffer.

8. The method of claim 1, wherein said translating step performed utilizing a user-level semantic routine.

9. A processor which has a native instruction set and emulates instructions in a guest instruction set, said processor comprising:

guest instruction storage that stores guest instruction from a guest instruction set, wherein said series includes a guest access instruction that indicates a guest logical address in guest address space;

semantic routine storage that stores a plurality of semantic routines of native instructions for emulating said series of guest instructions;

means, responsive to receipt of said guest memory access instruction for emulation, for translating said guest logical address into a guest real address and for thereafter translating said guest real address into a native physical address; and

means for executing a semantic routine that emulates said guest memory access instruction utilizing said native physical address.

10. The processor of claim 9, wherein said guest memory access instruction comprises one of a guest load instruction and a guest store instruction.

11. The processor of claim 9, wherein said guest memory access instruction comprises a guest instruction that initiates fetching of a guest instruction in said series from said associated memory.

12. The processor of claim 9, wherein said means for executing a semantic routine that emulates said guest memory access instruction comprises means for accessing said memory utilizing said native physical address.

13. The processor of claim 9, said means for translating said guest real address into a native physical address includes means for translating said guest real address into a native effective address and for then translating said native effective address into said native physical address.

14. The processor of claim 9, wherein:

said processor further comprising a translation lookaside buffer (TLB) containing entries utilized for address translation; and

said means for translating said guest logical address into a guest real address and for thereafter translating said guest read address into a native physical address includes:

means for determining if said translation lookaside buffer includes an entry that can be utilized to obtain said native physical address; and

means, responsive to a determination that said translation lookaside buffer contains an entry that can be utilized to obtain said native physical address, for translating said guest logical address into a guest real address and for thereafter translating said guest read address into a native physical address utilizing said translation lookaside buffer (TLB) entry.

15. The processor of claim 14, and further comprising:

means, responsive to a determination that said translation lookaside buffer does not contain an entry that can be utilized to obtain said native physical address, for creating an entry that can be utilized to obtain said native physical address in said translation lookaside buffer.

16. The processor of claim 9, wherein said means for translating includes means for executing a user-level semantic routine.

* * * * *

Received in the U.S.P.T.O.
In re Application of:
SOUMMYA MALLICK
Address Translation Buffer for Data Processing System Emulation Mode

ENCLOSED HEREWITH:

1. Subject Patent Application containing 36 pgs. of descrip., 6 pgs. of claims and 1 pg. of abstract
2. Eight sheets of drawings
3. Declaration and Power of Attorney
4. Assignment of inventors to IBM
5. Form PTO-1595
6. Transmittal (in duplicate)
7. Our return Postcard

IBM CORPORATION
Our File No. 0116AD-AA9-96-003
Mailed: 17 SEPTEMBER 1997
BFR:mb

09/22/97
08/934645
64930 U.S. PTO

2-6

PTO-103X
(Rev. 7-93)

FILING RECEIPT

UNITED STATES L ARTMENT OF COMMERCE
Patent and Trademark Office
ASSISTANT SECRETARY AND COMMISSIONER
OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

APPLICATION NUMBER	FILING DATE	GRP ART UNIT	FIL FEE REC'D	ATTORNEY DOCKET NO.	DRWGS	TOT CL	IND CL
08/934,645	09/22/97	2763	$770.00	AA9-96-003	8	16	2

ANDREW J DILLON
FELSMAN BRADLEY GUNTER & DILLON
SUITE 350 ARBORETUM POINT
9505 ARBORETUM BOULEVARD
AUSTIN TX 78759

Receipt is acknowledged of this patent application. It will be considered in its order and you will be notified as to the results of the examination. Be sure to provide the U.S. APPLICATION NUMBER, FILING DATE, NAME OF APPLICANT, and TITLE OF INVENTION when inquiring about this application. Fees transmitted by check or draft are subject to collection. Please verify the accuracy of the data presented on this receipt. If an error is noted on this Filing Receipt, please write to the Application Processing Division's Customer Correction Branch within 10 days of receipt. Please provide a copy of the Filing Receipt with the changes noted thereon.

Applicant(s) SOUMMYA MALLICK, AUSTIN, TX.

FOREIGN FILING LICENSE GRANTED 01/28/98
TITLE
ADDRESS TRANSLATION BUFFER FOR DATA PROCESSING SYSTEM EMULATION MODE

PRELIMINARY CLASS: 395

cc: B. Thomas

(see reverse)

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. :5,953,520
DATED :September 14, 1999
INVENTOR(S) :Soummya Mallick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Col.17, Line 10 please insert after "guest" --memory--.

Signed and Sealed this

Fifteenth Day of August, 2000

Attest:

Michelle Williams

Attesting Officer

Q. TODD DICKINSON

Director of Patents and Trademarks

Received in the U.S.P.T.O.
In re Application of:
SOUMMYA MALLICK
Address Translation Buffer For Data Processing
System Emulation Mode
ENCLOSED HEREWITH:
1. Request for Certificate of Correction;
2. Form PTO 1050 (Certificate of Correction); and
3. Our return postcard.
*Serial No. **08/934,645** Filed **09/22/97***
*Patent No. **5,953,520**; Issued **09/14/99***
IBM CORPORATION
*Our File No. **0116AD-AA9-96-003***
*Mailed: **January 28, 2000***
AJD:cpj

Staple Here Only !

PRINTER'S TRIM LINE

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. :5,953,520
DATED :September 14, 1999
INVENTOR(S) :Soummya Mallick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Page 17, Line 10 please insert after "guest" --**memory**--.

MAILING ADDRESS OF SENDER:

FELSMAN, BRADLEY, VADEN, GUNTER & DILLON, LLP
Suite 350, 7600B N. Capital of Tx. Hwy
Austin, Texas 78731

PATENT NO. 5,953,520

No. of add'l copies @ 50¢ per page

FORM PTO 1050 (Rev. 2-93)

Staple Here Only

PRINTER'S TRIM LINE

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. :5,953,520
DATED :September 14, 1999
INVENTOR(S) :Soummya Mallick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Page 17, Line 10 please insert after "guest" --**memory**--.

MAILING ADDRESS OF SENDER:
FELSMAN, BRADLEY, VADEN, GUNTER & DILLON, LLP
Suite 350, 7600B N. Capital of Tx. Hwy
Austin, Texas 78731

PATENT NO. 5,953,520

No. of add'l copi[es]
@ 50¢ per pa[ge]

FORM PTO 1050 (Rev. 2-93)

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

ATTORNEY DOCKET NO. **AA9-96-003**

In re Application of:	§	
	§	
SOUMMYA MALLICK	§	Examiner: **MOHAMED, A.**
	§	
Patent No. **5,953,520**	§	
	§	
Issued: **22 SEPTEMBER 1997**	§	Art Unit: **2763**
	§	
For: **ADDRESS TRANSLATION BUFFER FOR DATA PROCESSING SYSTEM EMULATION MODE**	§ § §	

REQUEST FOR CERTIFICATE OF CORRECTION

Hon. Commissioner of Patents
and Trademarks
Washington, D.C. 20231

Sir:

Applicant respectfully submits the enclosed Certificate of Correction, which corrects a typographical error made by the U.S. Patent and Trademark Office in the Letters Patent No. 5,953,520, issued September 14, 1999.

CERTIFICATE OF MAILING
37 CFR 1.8(A)

I hereby certify that this correspondence is being deposited with the United States Postal Service as First Class Mail in an envelope addressed to Commissioner of Patents and Trademarks, Washington, D. C. 20231.

1/28/00 ____________ Chris Mott ____________
Date ***Signature***

Received in the U.S.P.T.O.
In re Application of:
SOUMMYA MALLICK
Address Translation Buffer For Data Processing
System Emulation Mode
ENCLOSED HEREWITH:
1. Request for Certificate of Correction;
2. Form PTO 1050 (Certificate of Correction); and
3. Our return postcard.
*Serial No. **08/934,645** Filed **09/22/97***
*Patent No. **5,953,520**; Issued **09/14/99***
IBM CORPORATION
*Our File No. **0116AD-AA9-96-003***
*Mailed: **January 28, 2000***
AJD:cpj

cc: IBM

Received in the U.S.P.T.O.
In re Application of:
SOUMMYA MALLICK
Address Translation Buffer For Data Processing
System Emulation Mode
ENCLOSED HEREWITH:
1. Request for Certificate of Correction;
2. Form PTO 1050 (Certificate of Correction); and
3. Our return postcard.
*Serial No. **08/934,645** Filed **09/22/97***
*Patent No. **5,953,520**; Issued **09/14/99***
IBM CORPORATION
*Our File No. **0116AD-AA9-96-003***
*Mailed: **January 28, 2000***
AJD:cpj

OIPE JC42
FEB 03 2000
PATENT & TRADEMARK OFFICE

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

ATTORNEY DOCKET NO. **AA9-96-003**

In re Application of:	§	
	§	
SOUMMYA MALLICK	§	Examiner: **MOHAMED, A.**
	§	
Patent No. **5,953,520**	§	
	§	
Issued: **22 SEPTEMBER 1997**	§	Art Unit: **2763**
	§	
For: **ADDRESS TRANSLATION**	§	
BUFFER FOR DATA PROCESSING	§	
SYSTEM EMULATION MODE	§	

REQUEST FOR CERTIFICATE OF CORRECTION

Hon. Commissioner of Patents
and Trademarks
Washington, D.C. 20231

Sir:

Applicant respectfully submits the enclosed Certificate of Correction, which corrects a typographical error made by the U.S. Patent and Trademark Office in the Letters Patent No. 5,953,520, issued September 14, 1999.

CERTIFICATE OF MAILING
37 CFR 1.8(A)

I hereby certify that this correspondence is being deposited with the United States Postal Service as First Class Mail in an envelope addressed to Commissioner of Patents and Trademarks, Washington, D. C. 20231.

1/28/00 — ***Date***

Doris Motte — ***Signature***

No fee is required at this time; however, if a fee is required, please charge same to Deposit Account No. 09-0447.

Respectfully submitted,

Andrew J. Dillon
Registration No. 29,634
FELSMAN, BRADLEY, VADEN
GUNTER & DILLON, LLP
Suite 350, Lakewood on the Park
7600B North Capital of Texas Highway
Austin, Texas 78731
(512) 343-6116

ATTORNEY FOR APPLICANTS

Staple Here Only !

PRINTER'S TRIM LINE

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. :5,953,520
DATED :September 14, 1999
INVENTOR(S) :Soummya Mallick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Page 17, Line 10 please insert after "guest" --**memory**--.

MAILING ADDRESS OF SENDER:
FELSMAN, BRADLEY, VADEN, GUNTER & DILLON, LLP
Suite 350, 7600B N. Capital of Tx. Hwy
Austin, Texas 78731

PATENT NO. 5,953,520

No. of add'l copie @ 50¢ per pag

Staple Here Only !

PRINTER'S TRIM LINE

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. :5,953,520
DATED :September 14, 1999
INVENTOR(S) :Soummya Mallick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Page 17, Line 10 please insert after "guest" --memory--.

MAILING ADDRESS OF SENDER:
FELSMAN, BRADLEY, VADEN, GUNTER & DILLON, LLP
Suite 350, 7600B N. Capital of Tx. Hwy
Austin, Texas 78731

PATENT NO. 5,953,520

No. of add'l copie
@ 50¢ per pag

FORM PTO 1050 (Rev. 2-93)

EXHIBIT 16

US006009261A

United States Patent [19]

Scalzi et al.

[11] **Patent Number: 6,009,261**

[45] **Date of Patent: Dec. 28, 1999**

[54] **PREPROCESSING OF STORED TARGET ROUTINES FOR EMULATING INCOMPATIBLE INSTRUCTIONS ON A TARGET PROCESSOR**

[75] Inventors: **Casper Anthony Scalzi**, Poughkeepsie; **Eric Mark Schwarz**, Gardiner, both of N.Y.; **William John Starke**, Austin, Tex.; **James Robert Urquhart**, Fishkill; **Douglas Wayne Westcott**, Rhinebeck, both of N.Y.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **08/991,714**

[22] Filed: **Dec. 16, 1997**

[51] **Int. Cl.**[6] **G06F 9/455**

[52] **U.S. Cl.** **395/500.47**; 395/500.48

[58] **Field of Search** 257/51; 395/500.25, 395/500.35, 500.34, 500.21, 500.43, 500.47, 500.48, 500.49

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,587,612	5/1986	Fisk et al. .	
4,851,990	7/1989	Johnson et al. .	
5,077,657	12/1991	Cooper et al. .	
5,081,572	1/1992	Arnold .	
5,333,297	7/1994	Lemaire et al. .	
5,471,612	11/1995	Schlafly	707/104
5,488,729	1/1996	Vegesna et al. .	
5,574,927	11/1996	Scantlin	395/500.44
5,751,982	5/1998	Morley	712/209
5,857,094	1/1999	Nemirovsky	395/500.49
5,896,522	4/1999	Ward et al.	395/500.44
5,909,567	6/1999	Novak et al.	712/208

OTHER PUBLICATIONS

U.S. application No. 08/864,585, Greenspan et al., filed May 28, 1997.

U.S. application No. 08/864,402, Greenspan et al., filed May 28, 1997.

Annexstein et al., Acheiving Multilanguage Behavior in Bit-Serial SIMD Architectures Via Emulation, Feb. 1990, pp. 186–195.

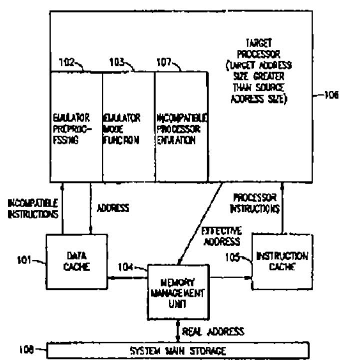
Hookway, Digital FX!32 Running 32-Bit X86 Applications on Alpha NT Mar. 1997, pp. 37–42.

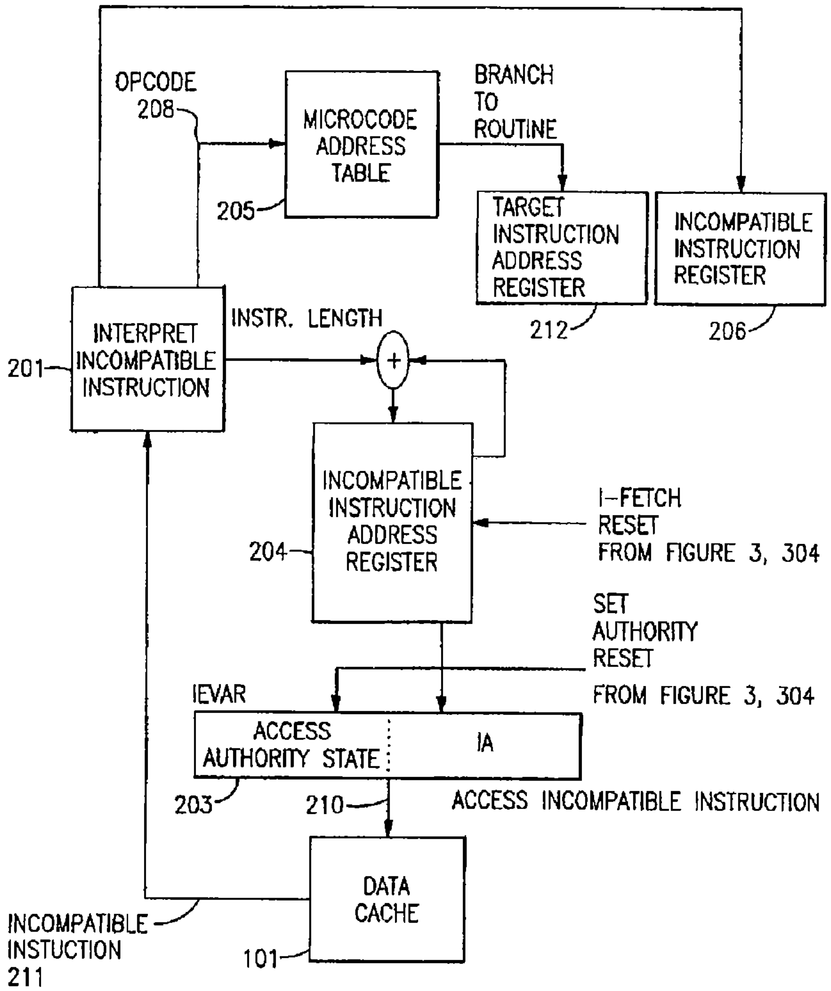
Primary Examiner—Kevin J. Teska
Assistant Examiner—Dan Fiul
Attorney, Agent, or Firm—Marc A. Ehrlich; Bernard M. Goldman

[57] **ABSTRACT**

Provides a program translation and execution method which stores target routines (for execution by a target processor) corresponding to incompatible instructions, interruptions and authorizations of an incompatible program written for execution on another computer system built to a computer architecture incompatible with the architecture of the target processor's computer system. The disclosed process allows the target processor to emulate incompatible acts expected in the operation of an incompatible program when the target processor itself is incapable of performing the emulated acts. Each of the instructions, interruptions and authorizations found in the incompatible programs has one or more corresponding target routines, any of which may need to be preprocessed before it can precisely emulate the execution results required by the incompatible architecture. Target routines (corresponding to the incompatible instruction instances in an incompatible program being emulated) are accessed, patched where necessary, and executed by a target processor to enable the target processor to precisely obtain the execution results of the emulated incompatible program. Before preprocessing, each target routine may not be able to provide identical execution results as required by the incompatible architecture, and the preprocessing may patch one or more of its target instructions to enable the target routine to perform the identical emulation execution of the corresponding incompatible instruction. The patching and other modifications to a target routine are done by one or more preprocessing instructions stored in the target routine.

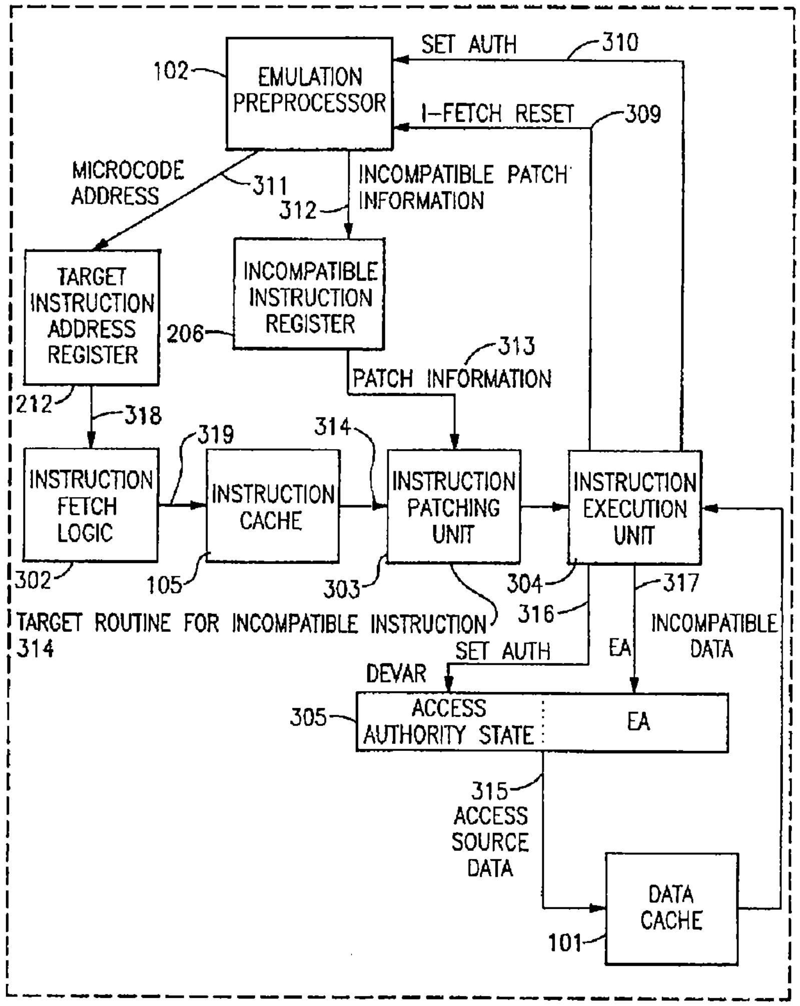
47 Claims, 13 Drawing Sheets





OPCODE
208
MICROCODE ADDRESS TABLE
205
BRANCH TO ROUTINE
TARGET INSTRUCTION ADDRESS REGISTER
212
INCOMPATIBLE INSTRUCTION REGISTER
206
INTERPRET INCOMPATIBLE INSTRUCTION
201
INSTR. LENGTH
+
INCOMPATIBLE INSTRUCTION ADDRESS REGISTER
204
I-FETCH RESET FROM FIGURE 3, 304
SET AUTHORITY RESET FROM FIGURE 3, 304
IEVAR
ACCESS AUTHORITY STATE
IA
203
210
ACCESS INCOMPATIBLE INSTRUCTION
DATA CACHE
101
INCOMPATIBLE INSTUCTION 211

FIG.2



EMULATION PREPROCESSOR
102
SET AUTH
310
I-FETCH RESET
309
MICROCODE ADDRESS
311
312
INCOMPATIBLE PATCH INFORMATION
TARGET INSTRUCTION ADDRESS REGISTER
INCOMPATIBLE INSTRUCTION REGISTER
206
313
PATCH INFORMATION
212
318
319
314
INSTRUCTION FETCH LOGIC
INSTRUCTION CACHE
INSTRUCTION PATCHING UNIT
INSTRUCTION EXECUTION UNIT
302
105
303
304
317
316
TARGET ROUTINE FOR INCOMPATIBLE INSTRUCTION
314
SET AUTH
EA
INCOMPATIBLE DATA
DEVAR
305
ACCESS AUTHORITY STATE
EA
315
ACCESS SOURCE DATA
DATA CACHE
101

FIG.3

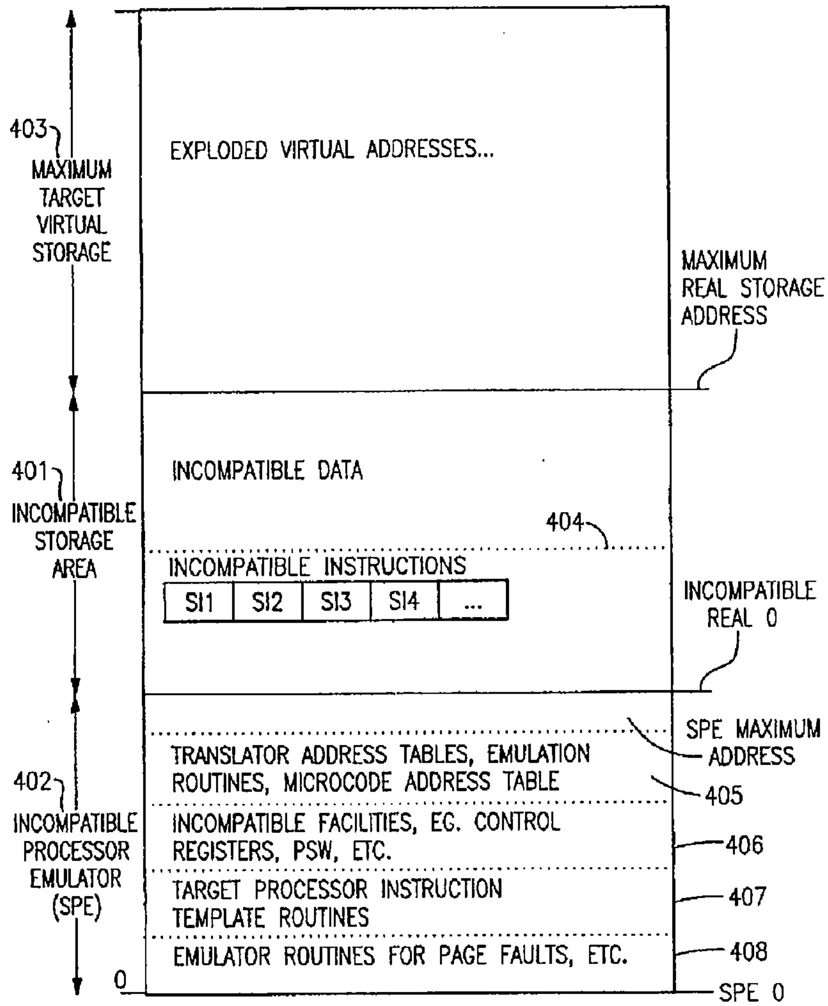
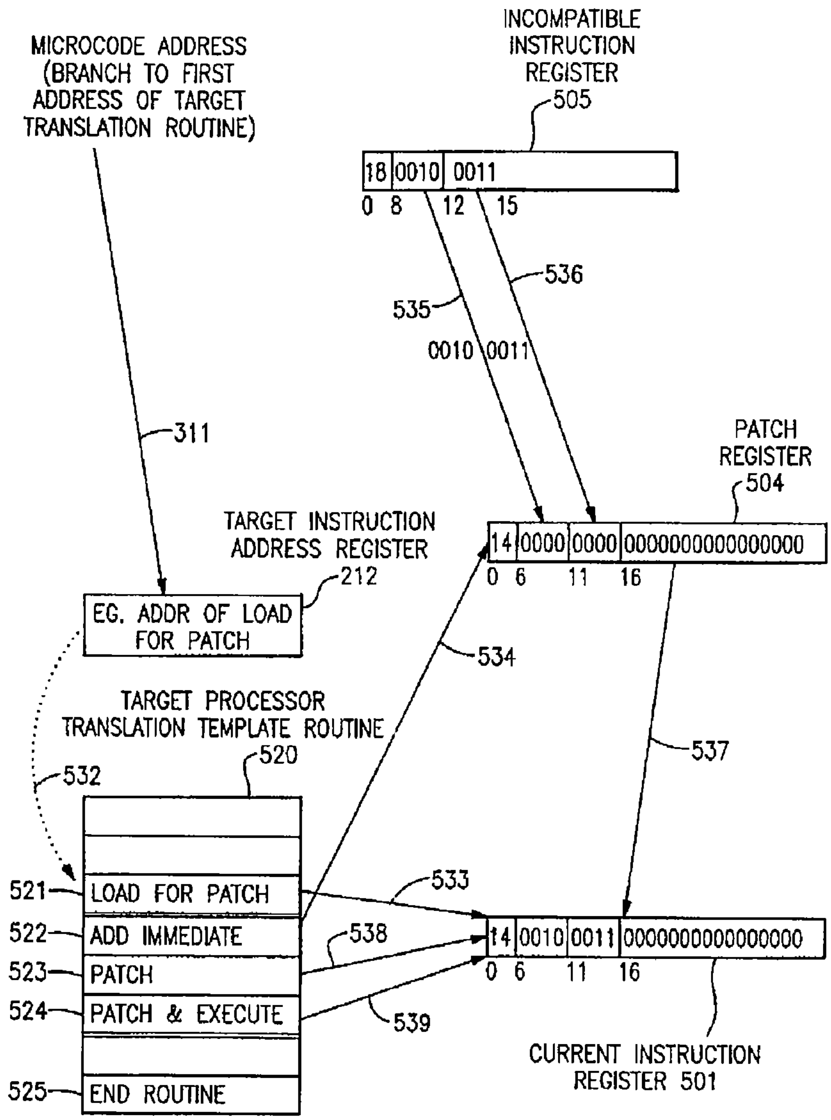

403
MAXIMUM TARGET VIRTUAL STORAGE
EXPLODED VIRTUAL ADDRESSES...
MAXIMUM REAL STORAGE ADDRESS
401
INCOMPATIBLE STORAGE AREA
INCOMPATIBLE DATA
404
INCOMPATIBLE INSTRUCTIONS
SI1
SI2
SI3
SI4
...
INCOMPATIBLE REAL 0
SPE MAXIMUM ADDRESS
402
INCOMPATIBLE PROCESSOR EMULATOR (SPE)
TRANSLATOR ADDRESS TABLES, EMULATION ROUTINES, MICROCODE ADDRESS TABLE
405
INCOMPATIBLE FACILITIES, EG. CONTROL REGISTERS, PSW, ETC.
406
TARGET PROCESSOR INSTRUCTION TEMPLATE ROUTINES
407
EMULATOR ROUTINES FOR PAGE FAULTS, ETC.
408
0
SPE 0

FIG.4



MICROCODE ADDRESS (BRANCH TO FIRST ADDRESS OF TARGET TRANSLATION ROUTINE)
INCOMPATIBLE INSTRUCTION REGISTER
505
18 0010 0011
0 8 12 15
536
535
0010 0011
311
PATCH REGISTER
504
TARGET INSTRUCTION ADDRESS REGISTER
212
14 0000 0000 0000000000000000
0 6 11 16
EG. ADDR OF LOAD FOR PATCH
534
TARGET PROCESSOR TRANSLATION TEMPLATE ROUTINE
520
537
532
521
LOAD FOR PATCH
533
522
ADD IMMEDIATE
538
14 0010 0011 0000000000000000
523
PATCH
0 6 11 16
524
PATCH & EXECUTE
539
CURRENT INSTRUCTION REGISTER 501
525
END ROUTINE

FIG.5

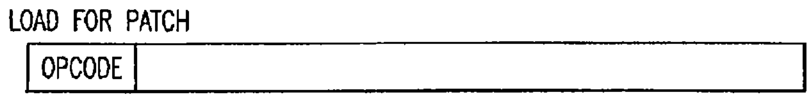


FIG.6

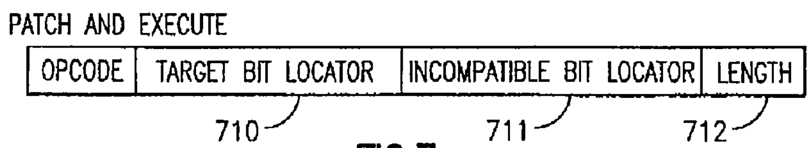


FIG.7

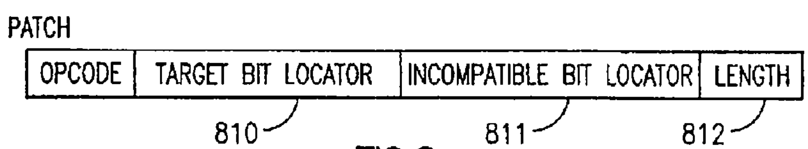


FIG.8

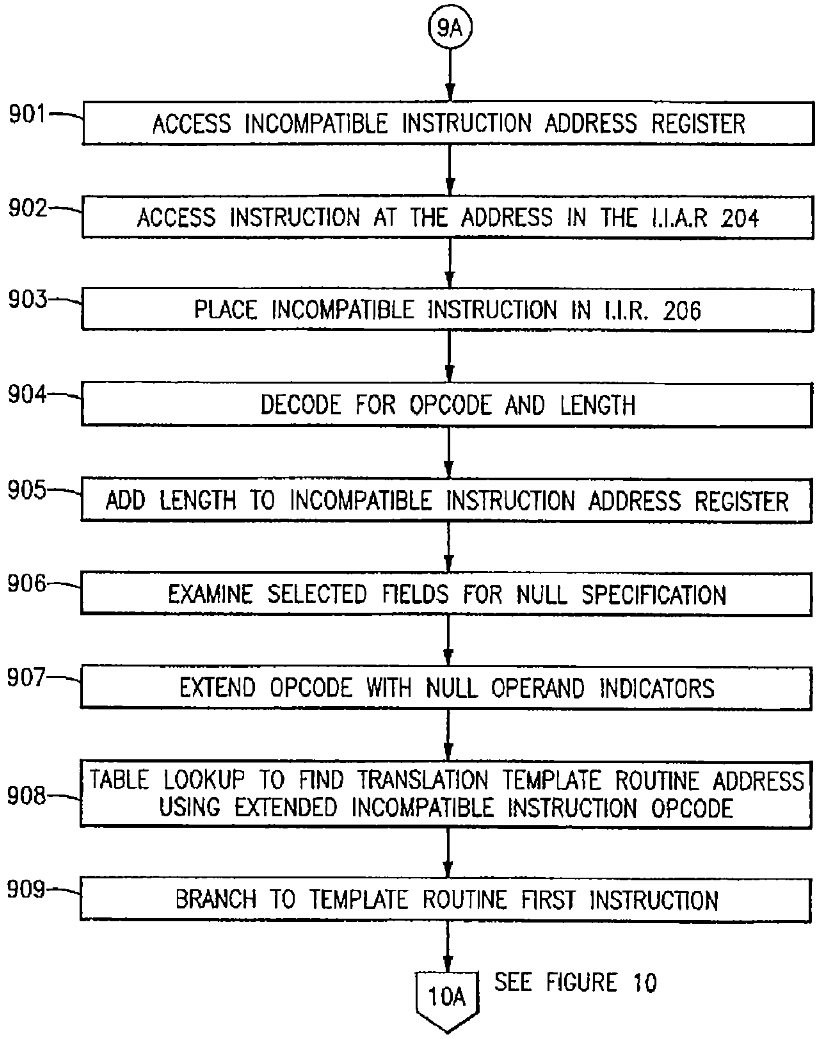

9A
901
ACCESS INCOMPATIBLE INSTRUCTION ADDRESS REGISTER
902
ACCESS INSTRUCTION AT THE ADDRESS IN THE I.I.A.R 204
903
PLACE INCOMPATIBLE INSTRUCTION IN I.I.R. 206
904
DECODE FOR OPCODE AND LENGTH
905
ADD LENGTH TO INCOMPATIBLE INSTRUCTION ADDRESS REGISTER
906
EXAMINE SELECTED FIELDS FOR NULL SPECIFICATION
907
EXTEND OPCODE WITH NULL OPERAND INDICATORS
908
TABLE LOOKUP TO FIND TRANSLATION TEMPLATE ROUTINE ADDRESS USING EXTENDED INCOMPATIBLE INSTRUCTION OPCODE
909
BRANCH TO TEMPLATE ROUTINE FIRST INSTRUCTION
10A
SEE FIGURE 10

FIG.9

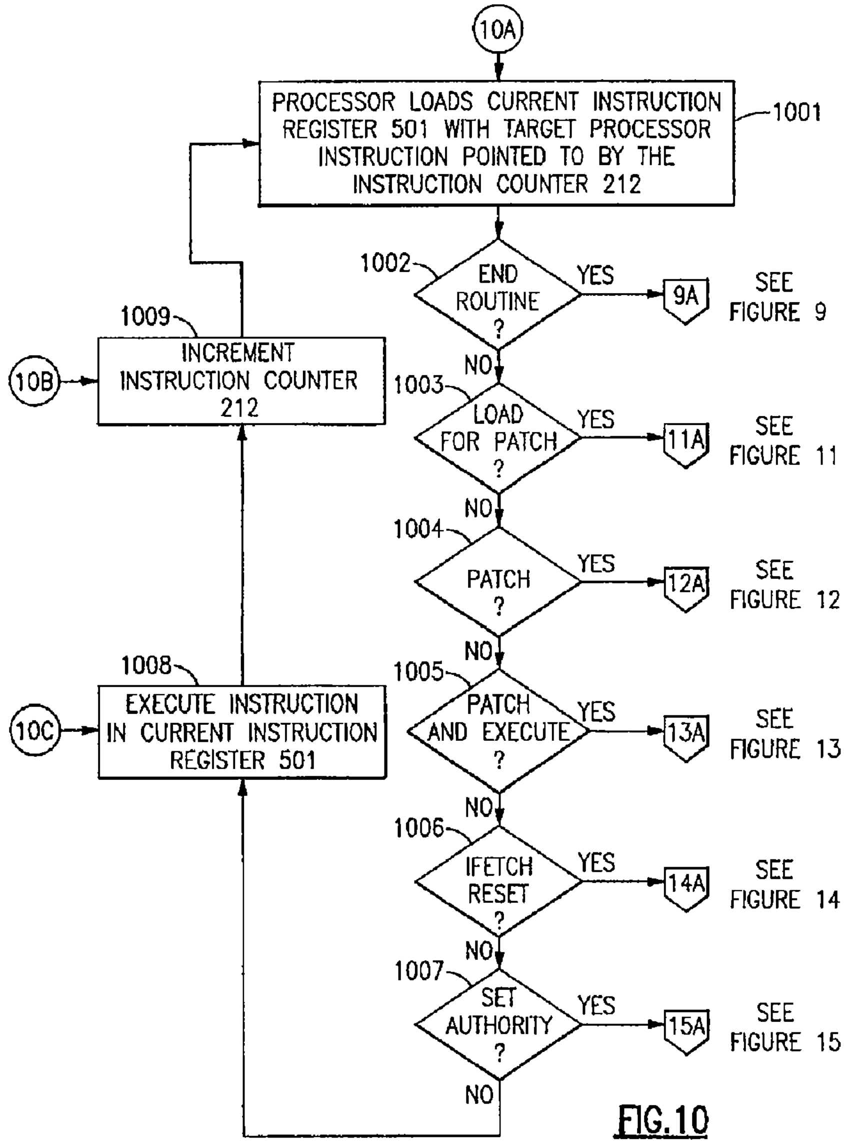

10A
PROCESSOR LOADS CURRENT INSTRUCTION REGISTER 501 WITH TARGET PROCESSOR INSTRUCTION POINTED TO BY THE INSTRUCTION COUNTER 212
1001
1002
END ROUTINE ?
YES
9A
SEE FIGURE 9
NO
1003
LOAD FOR PATCH ?
YES
11A
SEE FIGURE 11
NO
1004
PATCH ?
YES
12A
SEE FIGURE 12
NO
1005
PATCH AND EXECUTE ?
YES
13A
SEE FIGURE 13
NO
1006
IFETCH RESET ?
YES
14A
SEE FIGURE 14
NO
1007
SET AUTHORITY ?
YES
15A
SEE FIGURE 15
NO
1009
10B
INCREMENT INSTRUCTION COUNTER 212
1008
10C
EXECUTE INSTRUCTION IN CURRENT INSTRUCTION REGISTER 501

FIG.10

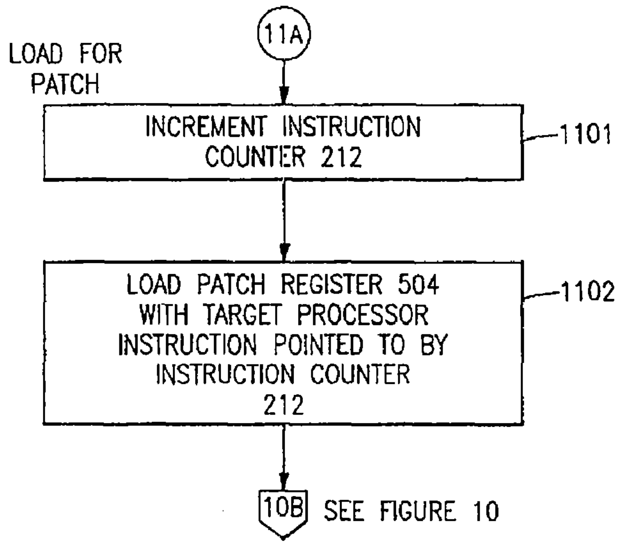

11A
LOAD FOR PATCH
INCREMENT INSTRUCTION COUNTER 212
1101
LOAD PATCH REGISTER 504 WITH TARGET PROCESSOR INSTRUCTION POINTED TO BY INSTRUCTION COUNTER 212
1102
10B
SEE FIGURE 10

FIG.11

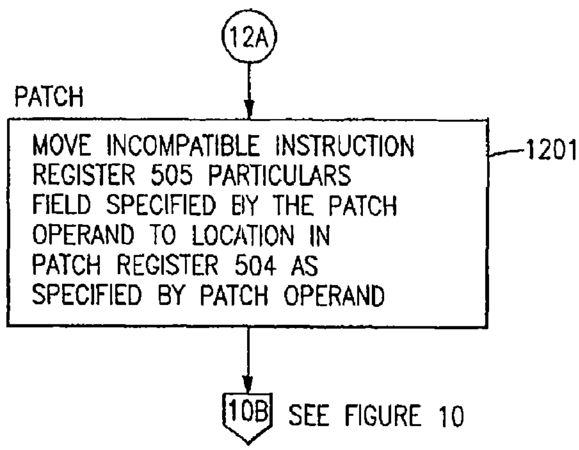

12A
PATCH
MOVE INCOMPATIBLE INSTRUCTION REGISTER 505 PARTICULARS FIELD SPECIFIED BY THE PATCH OPERAND TO LOCATION IN PATCH REGISTER 504 AS SPECIFIED BY PATCH OPERAND
1201
10B
SEE FIGURE 10

FIG.12

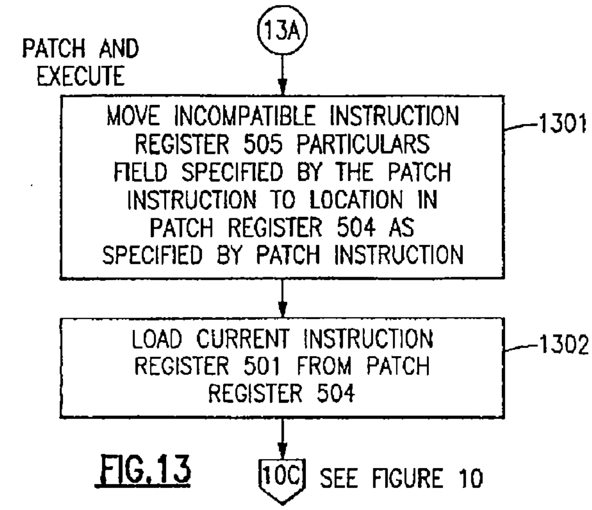


FIG.13

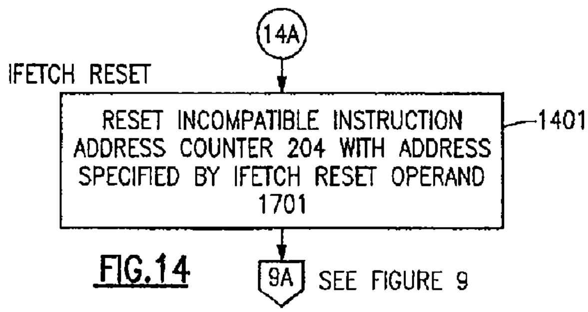


FIG.14

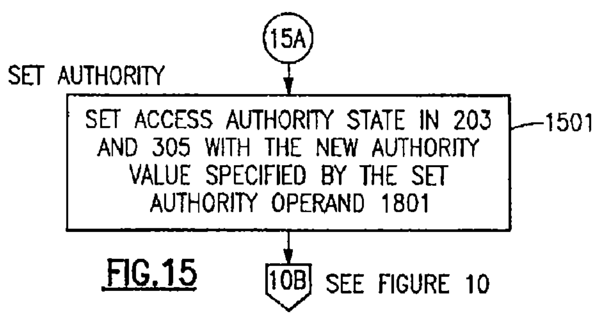


FIG.15

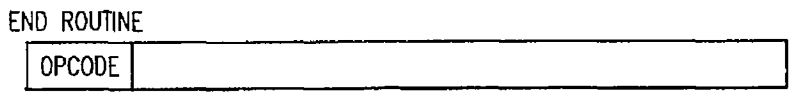


FIG.16

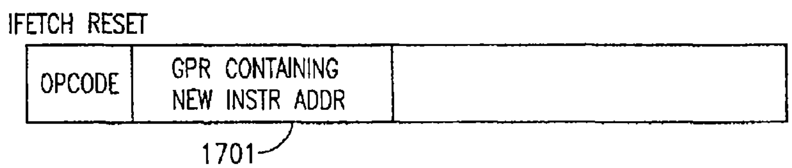


FIG.17

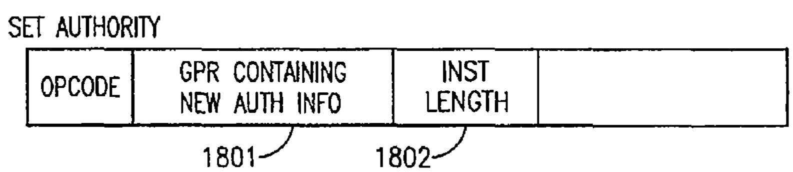


FIG.18

INCOMPATIBLE INSTRUCTION: L R,D(X,B)	
LOAD FOR PATCH	
ADD	
PATCH	UPDATE ADD WITH X
PATCH AND EXECUTE	UPDATE ADD WITH B AND ADD X + B
RLDIMI	UPDATE DEVAR WITH X+B
LOAD FOR PATCH	
LWZ	
PATCH	UPDATE LWZ WITH D
PATCH AND EXECUTE	UPDATE LWZ WITH R AND LOAD R
END ROUTINE	

RISC INSTRUCTIONS:

ADD=ADD INSTRUCTION

RLDIMI=ROTATE LEFT DOUBLE IMMEDIATE THEN MASK INSERT

LWZ=LOAD WORD AND ZERO

FIG.19

INCOMPATIBLE INSTRUCTION: L R,D(0,B)

RLDIMI	UPDATE DEVAR WITH B
LOAD FOR PATCH	
LWZ	
PATCH	UPDATE LWZ WITH D
PATCH AND EXECUTE	UPDATE LWZ WITH R AND LOAD R
END ROUTINE	

RISC INSTRUCTIONS:

RLDIMI=ROTATE LEFT DOUBLE IMMEDIATE THEN MASK INSERT

LWZ=LOAD WORD AND ZERO

FIG.20

PREPROCESSING OF STORED TARGET ROUTINES FOR EMULATING INCOMPATIBLE INSTRUCTIONS ON A TARGET PROCESSOR

INTRODUCTION

The subject invention generally relates to enabling programs written for execution on a computer system built to one type of computer architecture to be executed on a target computer system built to a target architecture different from the architecture of the emulated computer system. More specifically, the subject invention provides methods and means for a target computer system to emulate acts expected in the operation of an emulated program when the target computer system is itself incapable of performing the emulated acts.

INCORPORATION BY REFERENCE

This application incorporates by reference the entire content of each of the two following applications:

1) U.S. Pat. No. 5,560,013 (PO994067) entitled "Method of using a Target Processor to Execute Programs of a Source Architecture that uses Multiple Address Spaces", filed on Sep. 24, 1996 by C. A. Scalzi and W. J. Starke.

2) U.S. Pat. No. 5,577,231 (PO994041) entitled "Storage Access Authorization Controls in a Computer System Using Dynamic Address Translation of Large Addresses", filed on Nov. 19, 1996 by C. A. Scalzi and W. J. Starke.

The above incorporated patents teach program emulation methods and mechanisms, some of which are used in the subject invention. The U.S. Pat. No. 5,560,013 and U.S. Pat. No. 5,577,231 patent teach emulation methods using a target architecture having a much larger virtual address size than found in the emulated (source) architecture, and they trade virtual address space in the target system for emulated operational characteristics in the emulated architecture which may be alien to, and unrecognizable by, the target architecture, such as authority controls found only in the emulated architecture and not found in the target architecture. The embodiments in these incorporated patents emulate the complex IBM S/390 architecture, which contains elaborate storage access authorization mechanisms, for allowing an operating system to provide a programming environment which enforces strong integrity characteristics with regard to programs, and allows establishment of restricted data access domains. The more complex of such mechanisms generally have no counterpart in simpler target processor architectures, such as the Power PC and Intel CISC architectures. To provide the total operational execution environment of a complex incompatible architecture, the embodiments in these patents teach how the S/390 access authorization mechanisms may be emulated in a target machine not having these mechanisms. In more detail, these S/390 access authorization mechanisms include real storage access keys, storage access authority granted to programs, storage fetch-only authority, storage write authority, and address space access authority. These mechanisms must be properly emulated in a target processor to support a real S/390 operating scenario capable of executing S/390 application programs, in which the scenario includes OS/390 software, middleware programming, and the application programming.

BACKGROUND

U.S. Pat. No. 4,587,612 (Fisk, et al.) describes a system for accelerated instruction mapping of a program in the

architecture of one system, called the source, to instructions of a different system, called the target. The translation is done by an independent processor provided for the purpose and executing concurrently, and in tandem with, the target processor. The processor doing the translation is called an Emulation Assist Processor (EAP). That patent teaches the substitution of specific register values, and immediate displacement values, from source machine instruction into target machine instructions which, either singly or as a sequence of more than one, perform the function of the source machine. The technique described cannot be used to provide a total operational S/390 program execution environment. It does not describe a method to access emulated source main storage. Complex source machine architecture, e.g. S/390, often contains elaborate storage access authorization mechanisms for allowing an operating system to provide a programming environment which has strong integrity characteristics with regard to other programs, and allowing establishment of restricted data access domains. Such mechanisms generally have no counterpart in simpler target processors. To provide the total operational execution environment of a complex source machine, such access authorization mechanisms must be emulated in the target machine.

Examples of such architected mechanisms in a S/390 embodiment are real storage access keys, storage access authority granted to programs, storage fetch-only authority, storage write authority, and address space access authority. These mechanisms must be properly emulated in a target processor to support a real S/390 operating scenario including OS/390 software, middleware programming, and applications programming.

U.S. Pat. No. 5,577,231 (Scalzi and Starke) describes an efficient method of providing such access authorization mechanisms for a source machine with, for example, 31-bit addressing architecture on a simpler target machine with a larger size addressing architecture, e.g. one with 64-bit addresses. The additional address bits provide the capability of emulating S/390 access authority mechanisms as part of the target machine virtual address. The target machine instructions executed to provide the function of source machine instructions do not require 64 bits in order to access the emulated virtual storage or main storage of the source machine. For example, in S/390 source machines, only 31 bits of the target machine address are required as an offset or address into the emulated S/390 absolute main storage, and the basic addresses within an address space are only 31 bits. Excess high-order bits are used instead to emulate access under the various states of the source machine storage access controls. The source address, extended on the left by an access authority state, is called a target processor exploded virtual address. The high-order bits of the exploded virtual address represent the access authority of the source program making the storage request. On first reference to a particular location under a specific setting of the source access controls, a target page fault will occur. The emulator kernel, either a program or microcode, on the target machine will determine the validity of the access in resolving the page fault. If the access is invalid, the architected response for the invalid condition encountered is reflected to the appropriate source program as specified in the source machine architecture. If the access is valid, a Page Table Entry (PTE) is established for the exploded address so that the target page frame emulating the accessed source storage page may be accessed at full target machine processor speed without further intervention, as long as the PTE remains established for that target virtual page. The target processor

exploded virtual address in the PTE represents two things in composite: the source machine storage address (either virtual or real), and the storage access authority under which the access was made and found valid. The instruction that caused the page fault is re-executed. Its and future accesses to the same source location under the same access authority occur at full target machine performance through the established PTE. If the same source storage location is then accessed from a different source storage access authority state, the target machine will use a different target virtual address to access the location. The exploded address used will be different than on the earlier access. In a S/390 embodiment, the low-order 31 bits will be the same as before, but the high-order bits will reflect a different source access authority state; for example, a different PSW storage access key. The target machine will not resolve the second target virtual address used with the PTE established for the first access, since the target processor address is different. The first time this second address is used, a page fault occurs, and the source access with the new authority state can be validated by the emulator kernel. If that access is also valid, a second, different PTE is established for the access from the second authority state.

U.S. Pat. No. 5,560,013 (Scalzi and Starke) describes an efficient programming method to apply the inventive concepts of U.S. Pat. No. 5,577,231 in the emulation of a complete program operational environment of a computing system with complex access authority mechanisms in its architecture; for example, to emulate a total S/390 program operating environment, including an operating system, middleware, and application programs. The machine language instructions of an alien source computer are translated, as encountered, in the emulated execution of a source computer program on a different target computer. A translation exists for each possible source processor instruction type, and consists of one or more target machine instructions that together will provide the function of the source machine instruction. As each source instruction is encountered, the target translation is accessed, and the target instructions of the translation are modified so that they are made to reflect the specifics of the particular instance of the source instruction, e.g. registers, addresses, displacements, etc. After this modification, the instructions of the translation are saved in a target storage area so that the instruction translation is not required on each execution of the same source instruction during the complete source program emulated execution. As long as the translation remains valid, the existing translation can be executed directly in place of the source instruction. A directory is maintained with an entry for each possible source machine instruction location. This directory indicates whether or not a valid translation already exists for each possible instruction instance. To execute a source instruction during emulated execution of the source program, a branch is taken to the directory entry for the source instruction. If a translation already exists, the directory entry contains a branch instruction to it. Otherwise, the directory entry branches to the instruction translation routine to create the necessary target translation for the source instruction.

SUMMARY OF THE INVENTION

The subject invention provides a program translation and execution system which allows the processor to execute an incompatible program coded to a different computer architecture. The design allows an emulation that avoids the large target storage overhead of the solution provided by U.S. Pat. No. 5,560,013. By doing the instruction translation in a dynamic manner each time an incompatible instruction is encountered, it is unnecessary to save the translations for future use, saving the storage required to preserve these in target storage. The invention provides new functions in emulation mode that are used to personalize target instructions with the characteristics of the incompatible instruction they are to emulate. Also, a register is allocated to hold the incompatible instruction and be an operand location for these new instructions. This allows a standard microprocessor to be used to perform target machine instructions that provide the same results as the incompatible instructions being executed by this emulation technique. Preprocessing is performed in emulation mode to interpret the incompatible instruction to be executed and to select the proper target translation routine to perform its function. This preprocessing can be provided by means of a hardware preprocessor, or be implemented in microcode, or simply be performed by a program executing on the target processor. In the embodiment shown herein, it is a program.

In the emulation operating mode the target instructions to be executed are determined by the preprocessing function which interprets each incompatible instruction to calculate which routine of target instructions must be executed next to continue execution of the incompatible program. In emulation mode, specialized functions are provided to manage and control the storage access authority of the program being emulated, and to reflect specifications of the incompatible instructions into target instructions performing their emulation. In the embodiment shown here, these specialized functions are called instructions, since that is one possible option for the implementation. An emulator, either implemented in microcode or as a program executing on the processor, maps unique incompatible architecture facilities to target machine facilities that are accessed by the target instructions in performing the emulation. The emulator also processes target machine interruptions which signal conditions that require it to validate and control the incompatible authority state and manage the incompatible instruction control flow changes. The target instructions of an incompatible instruction translation routine change the incompatible authority state as required by the incompatible instruction being emulated. They also modify the incompatible instruction address register, or instruction counter, when there is a change in the sequential execution of incompatible instructions.

The target instructions of a translation routine manage the unique architecture facilities of the incompatible program such as general purpose registers, control registers, instruction condition codes, program status words, etc. These are allocated to target machine registers or to target storage locations as appropriate, and processed in their mapped state by the target instructions of the translation routines.

The target instructions also perform any special processes of the incompatible architecture such as address arithmetic, or providing special incompatible instruction condition code indications.

In the microprocessor, the registers are larger than required by the storage addressing of the incompatible program. This allows the registers actually used to address the emulated incompatible storage also to reflect the current authority state of the incompatible program, implementing what is taught in U.S. Pat. No. 5,577,231. The target instructions of each incompatible instruction translation, in cooperation with the emulator, manage the high-order portion of registers used to access incompatible storage so as to reflect the current authority state of the incompatible program for each access being made. Target machine page

faults caused by target instructions in translation routines signal the emulator to validate the accesses to incompatible storage, using the emulated facilities of the incompatible architecture. These facilities are maintained by the target instructions of translation routines emulating incompatible instructions which change those facilities for the incompatible program.

The subject invention described here uses an emulation preprocessing function, which decodes the incompatible instructions and, for each, determines the location of a target processor translation template routine that performs the function of the incompatible instruction. A template routine is executed by the processor when the incompatible instruction is reached in the incompatible program instruction execution stream, as indicated by the address in the emulated incompatible instruction address register, or instruction counter.

Also, the particular specifications of each incompatible instruction are extracted from the incompatible instruction, e.g. register numbers specified, dynamically, each time an incompatible instruction is to be executed and used in target instructions of the template routine. It is not necessary to save previous translations to obtain good performance in the emulated execution. The large amount of target storage required to save the existing translations for future execution and for the incompatible instruction directory is saved by the dynamic translation of incompatible instructions as they are encountered during emulated execution.

The invention uses the methods of U.S. Pat. No. 5,577,231 to emulate incompatible machine access authority states in the emulation of incompatible machine program execution on a simpler execution target processor that does not contain hardware equivalents for the more complex incompatible authority mechanisms assumed and used by the incompatible programs.

The target processor provides and uses larger virtual addresses for its own instruction execution than are required by the incompatible instructions which must be performed. The high-order portion of each target machine virtual storage address which accesses incompatible storage is used to indicate the current state of incompatible access authority enablement, thus differentiating the different incompatible authority states from each other in target machine emulation of incompatible storage accesses during incompatible program execution.

The emulator mode facilities of the processor provide six new instructions that are found in the target instruction templates for incompatible instructions to be emulated. These instructions direct and control the emulator facilities as required in the performance of the source-equivalent target instructions. The Load for Patch, Patch, and Patch and Execute instructions are used to modify the instructions of a template in order to reflect the specifics of the incompatible instruction, such as register numbers or displacement amounts.

The End Routine instruction marks the end of a template and signals that a new incompatible instruction must be accessed and interpreted. Control is returned to the preprocessing to perform the access and interpretation, and the transfer to the translation template routine for the next incompatible instruction.

The Ifetch Reset instruction is executed in a template to signal that, as a result of executing the incompatible instruction emulated by the target instruction template routine, the content of the incompatible instruction counter, register 204, is to change. That is, the incompatible program is not to execute the next sequential instruction. Instead, a new value is placed into the emulated incompatible instruction address register to redirect the incompatible program's execution path. Control then passes to the preprocessing function. The preprocessing accesses the address in the incompatible instruction address register and uses it to fetch the next incompatible instruction, and then interprets the instruction, following the flow of the incompatible program as it develops.

The Set Authority instruction is executed in a template when, under the incompatible architecture rules, there has been a change to the storage authority of the incompatible program as a result of executing the incompatible instruction being emulated by the target template routine. Bits representing the new authority state are loaded into the high-order portion of the target registers used to access the incompatible instructions and the data those instructions access in the emulated incompatible storage. This causes all future accesses to incompatible storage to occur from the new access authority state, as represented by the high-order bits in the target virtual addresses actually used by the target instructions of the template routines to make the accesses.

Any of the six special emulation instructions may be implemented as microcoded instructions. Other implementations may provide them as macro-instructions and have them replaced during compilation by in-line code to perform their functions in the appropriate target translation routines. Alternatively, they can be compiled as program calls to routines in the emulator provided to perform these functions when called.

In brief, the preprocessing accesses the incompatible instructions, decodes each and determines the address of the target instruction template routine which emulates the incompatible instruction. It passes the particulars of the incompatible instruction instance to the processor in a incompatible instruction register, and transfers control to the first instruction of the translation routine for the incompatible instruction. The processor, in emulation mode, accesses and executes each instruction in the specified translation template routine in order. The template routine contains the target instructions whose execution emulates the incompatible instruction. Six special instructions are provided in emulation mode which are used in the template routines to modify specified target instructions in the template so that they reflect parameters of the incompatible instruction instance as actually coded, to signal the preprocessing of changes in its sequential execution flow, to establish a new authority state for accesses to incompatible storage for incompatible instructions or the data they process, and to signal the end of the target translation routine for the current incompatible instruction.

By use of the six special instructions, the template routines can fully emulate the incompatible instructions, without special target hardware in the processor to emulate incompatible instruction characteristics. The preprocessing function must be able to decode the format of the incompatible instructions enough to select a template routine. Different than U.S. Pat. No. 4,587,612 it need not include generation of the target instructions required to execute incompatible instructions and their modification to reflect the incompatible instruction instance. Instead, that function is provided by the programming of the template routines for the incompatible instruction executions, which use the special instructions, as needed, to direct normal execution of instructions by the processor at its rated performance.

In cases where the fields in incompatible instructions may be specified as null, separate template routines may be

designed that take advantage of the null specification. If an incompatible field need not be handled, the target template routine can contain fewer target instructions to be executed. For example, in S/390 certain instructions allow specification of two registers, base and index, and an immediate displacement field, the contents of which take part in operand effective address calculation. Frequently, only one register is actually specified, and often a zero displacement is coded.

The preprocessing function is designed to recognize such null specifications. Instead of just using the incompatible instruction opcode as the table lookup parameter to find the proper translation template routine address for the incompatible instruction, it extends the lookup field to include other bits, each representing an incompatible instruction field which is frequently coded as null. Each bit indicates whether the field was null or non-null.

Thus, the null cases can be distinguished from the non-null ones in the table lookup to find the address of the target routine to emulate the incompatible instruction. This yields faster emulation of incompatible instructions, some of whose fields are frequently coded with null values, by providing separate translation template routines for them when null values are found. These templates for incompatible instructions with null field specifications contain fewer target instructions to be executed, providing faster emulation of those incompatible instructions.

The invention efficiently adapts to emulate a system with an evolving architecture, such as S/390, in which new instructions are periodically added. For the described system, such new incompatible instructions can be handled by new incompatible instruction translation template routines. There is no hardware impact to handle the new incompatible instructions. Target instructions requiring no modification are executed directly from the template, requiring no control information to be examined in order to direct their handling. Also, since machine characteristics are contained in the target translation routines and the preprocessing function, the invention can be applied to the emulation of many incompatible architectures without affecting the target processor hardware.

All accesses to the target system storage in behalf of the incompatible program are made with target virtual addresses larger than the incompatible program addresses they emulate, with the high-order bits representing the particular access authority state of the incompatible program at the time of access. This is done for the accesses of the incompatible instructions by the preprocessing code, and for data accesses to incompatible storage made by target instructions emulating data accesses by the incompatible instructions.

BRIEF DESCRIPTION OF DRAWINGS

The subject matter which is regarded as the invention is particularly pointed out and distinctly claimed in the claims at the conclusion of the specification. The foregoing and other objects, features and advantages of the invention will be apparent from the following detailed description taken in conjunction with the accompanying drawings which are:

FIG. 1 illustrates the main system elements in which the invention operates.

FIG. 2 depicts the elements of the logic of the emulation preprocessing of an incompatible program.

FIG. 3 depicts the elements of processor execution in emulation mode.

FIG. 4 shows the target machine storage mapping.

FIG. 5 depicts the operation of the Patch Unit in emulation mode operation of the processor.

FIG. 6 depicts the format of the Load For Patch instruction.

FIG. 7 depicts the format of Patch and Execute instruction.

FIG. 8 depicts the format of the Patch instruction.

FIG. 9 depicts the logic of incompatible instruction interpretation.

FIG. 10 depicts the processor emulation mode execution logic.

FIG. 11 depicts the processor execution logic for a Load for Patch instruction execution.

FIG. 12 depicts the processor execution logic for a Patch instruction execution.

FIG. 13 depicts the processor execution logic for a Patch and Execute instruction execution.

FIG. 14 depicts the processor execution logic for a Ifetch Reset instruction execution.

FIG. 15 depicts the processor execution logic for a Set Authority instruction execution.

FIG. 16 depicts the format of the End Routine instruction.

FIG. 17 depicts the format of the Ifetch Reset instruction.

FIG. 18 depicts the format of the Set Authority instruction.

FIG. 19 depicts the translation template for an incompatible load instruction which specifies a base register, an index register and a displacement.

FIG. 20 depicts the translation template for an incompatible load instruction which specifies a base register and a displacement, but no index register.

DESCRIPTION OF THE DETAILED EMBODIMENT

Elements of the invention are depicted in FIG. 1 in a target processor 106, which may be a standard microprocessor using RISC or CISC architecture, for example. The target processor also contains preprocessing function 102, which controls an incompatible instruction address register, accesses the incompatible instruction next to be executed, interprets its opcode, tests selected fields in the incompatible instruction for null specification, prepares a table lookup value from the incompatible instruction opcode extended by null field specification summary bits, and uses the table lookup value to find the address in the target memory of a corresponding target routine that performs the function of the incompatible incompatible instruction to be executed. An emulator mode function 103 processes the target instructions of the selected target translation routine to reflect specifics of the incompatible instruction instance to be emulated by moving any necessary fields from the incompatible instruction to appropriate target instruction(s) in the target routine before each of its target instructions is executed. This process of modifying the target instructions in a target routine is herein termed "patching" and is performed by three patching instructions, which may be hardware-implemented, or provided in microcode-implemented, or program-implemented in various types of embodiments available for this invention. The preferred embodiment described in detail herein uses a microcoded implementation in the target processor. The microcode is software which is protected from being changed by user software executing in the target processor.

Incompatible processor emulation 107 executes each target instruction in the target routine after any patching has

been done to the target instruction, so that when execution of the target routine is completed it obtains the same effect as the execution of the incompatible instruction would obtain if executed on a machine built to the architecture of the incompatible program being emulated. Incompatible processor emulation 107 includes of a software which manages the mapping of incompatible architecture facilities to facilities of the target machine for purposes of the emulation, and handles page fault processing to validate incompatible storage access authority states and establish these in the target system for efficient future accesses without such validation.

The preprocessor 107 accesses incompatible instructions as data from target system main storage 108 through a data cache 101 which is shared with the target processor 106. The target processor 106 accesses its instructions for execution from an instruction cache 105. Memory management unit 104 handles cache misses and cache castouts (stores) to system main storage 108.

FIG. 2 shows the operation of the incompatible instruction preprocessing function 102. It contains an incompatible instruction address counter, register 204, used to access the incompatible instructions from the portion of target system storage assigned as incompatible storage area 401 (shown in FIG. 4). Incompatible instructions are fetched from there by the preprocessing function 102 for analysis of the operation codes and, possibly, other fields as well. In emulation mode, each incompatible instruction instance is translated to a set of equivalent target instructions in a corresponding target translation routine for execution in the target processor. There is at least one target processor translation routine corresponding to each incompatible instruction type, which routine must be modified to reflect specifications of the incompatible instruction instance.

For example, in emulating S/390 incompatible instructions (which are herein presumed to be incompatible with the target processor), the register numbers and displacement amounts specified in the S/390 instruction must be reflected in the appropriate target instructions in the corresponding target routine which emulates the S/390 instruction. For this reason, a target routine is herein called a "translation template". The preprocessing function 107 (shown in detail in FIG. 2) determines the address in the target processor of the corresponding translation template for each incompatible instruction instance encountered during the emulated execution of the incompatible program, and this address branches the target machine execution to the first instruction of the selected routine.

In complex incompatible machine instructions (exemplified by the S/390 architecture), specifications in addition to the operation code may be used to determine what template routine is to be executed, such that an incompatible instruction type may have more than one corresponding template routine (hereafter called a "template"). The template is selected in table 205 for execution, and its selection is determined from the overall specifications of the incompatible instruction instance. The parameters in each incompatible instruction used for selection are preplanned and precoded, and used by preprocessing 102 to determine which translation template is to be used for each particular incompatible instruction instance. For example, certain incompatible instruction fields may be tested for a null specification in the instance to be emulated, and a different corresponding routine may be selected when a tested field is null in the current instance. The advantage of null field testing is that the corresponding template may have fewer target instructions to be executed in emulating that incompatible instruction instance, when compared to using a corresponding template which supports all available values for that field. This advantage becomes important when the null field condition is frequently used in the incompatible instructions being emulated.

To find an associated target processor translation template routine, a table lookup operation in table 205 is performed using the incompatible instruction operation code plus any other qualifier(s) chosen for testing for this purpose in the incompatible instruction instance, that together uniquely identify the required translation template routine to use in order to perform the currently executing instance of the incompatible instruction. This lookup selects a translation template that provides a specific target machine equivalent routine to perform the exact same functions as a native processor would for the given incompatible instruction. The target processor translation templates for all incompatible instructions are kept in area 407 in protected emulator storage (see FIG. 4).

In FIG. 3, the preprocessor 102 loads the incompatible instruction to be executed into the incompatible instruction register 206 where it is available as a source of patch information for an emulation mode patching operation. The particulars from the incompatible instruction instance are called patch information because they are used to modify or "patch" the target instructions of the translation routine which will perform the functions of the incompatible instruction. The patch information may be the incompatible instruction text itself as depicted in this embodiment; however, other variations using information extracted from the instruction instance may also be used.

When the incompatible program control flow changes from its normal execution flow due to a branch or an interruption, the emulated incompatible instruction address register is loaded with the new location for target instruction execution to reflect the change, and this affects the instruction preprocessing to be performed next.

In FIG. 2 the incompatible instruction address of the next incompatible instruction to be executed is read out from the instruction counter, register 204, and the incompatible instruction fetched from incompatible storage area 401 in the target system storage. The access is performed through the Instruction Exploded Virtual Address Register IEVAR 203. The high-order part of the IEVAR contains a value representing the current state of the program authority for accesses to incompatible storage for instruction accesses. The incompatible instruction is interpreted based on the incompatible architecture instruction format. The opcode obtained is used, possibly in combination with bits representing other aspects of the coded incompatible instruction instance, to form a search parameter for a table lookup operation in the Microcode Address Table 205 to obtain the address of the corresponding target processor translation routine that provides the same function as the incompatible instruction to be emulated.

This process is performed by incompatible instruction interpreter 201. The length of the incompatible instruction is used to update the incompatible instruction address register 204 for accessing the next incompatible instruction. The specified particulars of the incompatible instruction instance are provided by the interpreter for later patching operations by its loading the current incompatible instruction (being emulated) into a incompatible instruction register 206. The template address obtained from the Microcode Address Table is used to branch to the corresponding target routine for execution, and the tempate address is placed in target instruction address register 212.

When the emulation function 107 (see FIG. 1) in the target processor determines a non-sequentiality in incompatible instruction flow, it resets the incompatible instruction address register 204 with the new instruction counter value. When the execution of the incompatible program causes a change in the access authority for instruction access by the incompatible program, an Instruction Exploded Virtual Address Register (IEVAR) 203 is set by the emulator operation, which changes the high-order address part of IEVAR 203. Storage accesses in the target storage for incompatible instructions are performed through use of IEVAR 203, in which the current access authority state of the incompatible program for instruction fetching is reflected in the high-order bits of the exploded target virtual address used to access the incompatible instructions.

FIG. 3 depicts the emulator functions 103 performed in the target processor. The preprocessor function 102 uses transfer path 312 to transfer the selected patch information to the incompatible instruction register 206, which receives the incompatible instruction and provides the necessary patch information to be used to modify target instructions in the corresponding template routine which is to emulate the incompatible instruction The target instruction address register 212 determines the target processor's execution path. Then the preprocessor 102 branches to the selected target translation routine which is to perform the functions of the incompatible instruction for each incompatible instruction, in the order in which the incompatible program execution dictates that its instructions should be executed. The instruction fetch logic 302 of the target processor reads the content of the target instruction address register 212 on line 318 and requests the instruction from storage on line 319.

The instruction cache 105 supplies the required instruction on line 314. In box 303 the target instructions of the routine are then each modified, or patched, as necessary for the incompatible instruction type, to reflect the particulars of the incompatible instruction instance they are to perform, e.g. particular registers or displacements specified. Detailed operations of this process are illustrated in FIG. 5. The target instructions are then executed by the processor instruction execution unit 304.

If any of the target instructions require no patching, they pass through directly to the instruction execution unit 304 for execution without a patching step in 303. Where the execution results in a change in the sequential flow of the incompatible program, the incompatible instruction address register 204 is changed by instructions in the translation routine in order to redirect the operations of the preprocessing function, as depicted by line 309. Where the execution results in a change to the storage access authority of the incompatible program, the new authority is set into the high-order part of the IEVAR 203 by instructions of the translation routine, depicted by line 310, thus affecting accesses for incompatible instructions made by the preprocessor.

During target instruction execution emulating incompatible instructions, accesses to an incompatible instruction storage data area in the target system storage for data occur by means of the Data Exploded Virtual Address Register (DEVAR) 305. The incompatible effective address (EA) is sent to the low-order part of DEVAR 305, as illustrated by line 317 in FIG. 3. The high-order part of DEVAR 305, which normally remains unchanged for thousands of incompatible instructions at a time, reflects the current data access authority of the incompatible program under incompatible architecture. Thus, as explained earlier for incompatible instruction addresses (used to access instructions in an emulated incompatible storage area 401 in the target system storage), the high-order part of the target address represents the incompatible machine access authority state (not supported in the target processor hardware) of the program making the data access. When the access authority of the incompatible program changes as a result of its own execution, instructions of the translation routine for the incompatible machine authority-changing instruction will change the high-order part of the DEVAR 305.

A simple example for a S/390 incompatible program would be an access of data in another address space which would require part of the high-order part of the target exploded address to be changed to reflect the new virtual address space of the reference, forcing the incompatible program's authority to access the address space to be validated, if it had not yet been checked. The first access to an incompatible storage location under a new incompatible program authority state will result in a target page-fault, causing the microcoded or programmed emulator to check the validity of access and, when valid, establish a target page-table entry (PTE) for the full target location address, including the incompatible access state. All following accesses to the same page will occur at full target processor speed through the established PTE, accessed from the data cache 101. However, a reference to the same page under a different authority state will cause a target page-fault, allowing the emulator program to check the validity of the reference under the new authority state. Should the new reference be valid, another, different, target processor PTE is established for the second target virtual address which accesses the same emulated location. Of course, both PTE's resolve the different target virtual addresses to the same target real location, that of the accessed location. When the emulated execution of the incompatible instruction results in a change in the established data accessing authority of the incompatible program, the DEVAR is changed to reflect the change. This is indicated by the authority reset line 316 between 304 and 305.

Accesses that are found invalid under incompatible architecture rules are reflected to the incompatible programming environment in accordance with the incompatible architecture for such violations. The use of a target virtual address larger than the incompatible architecture address to represent incompatible machine access authority states is fully described in U.S. Pat. No. 5,577,231.

In FIG. 4, a contiguous portion of the target real storage is assigned as the incompatible storage area 401 on a sequential address basis. The incompatible storage area begins at the target address labeled INCOMPATIBLE REAL 0 in FIG. 4, and extends to the last storage location required to hold the maximum address in the incompatible storage area, which is the last target location in the area 401. To find an incompatible storage real address in the target storage the target real address of incompatible real storage location zero must be added to it. To provide incompatible program execution, the preprocessor accesses the incompatible instruction addressed by the incompatible instruction address register 204 at its equivalent target storage address. Access of the incompatible instruction is made using a target machine exploded virtual address reflecting the current storage access authority state of the incompatible program for instruction fetching. The overall emulation process interprets that instruction as an incompatible instruction and decomposes it in accordance with the incompatible architecture. For example, where S/390 represents the incompatible architecture, the instruction's operation code, base and index register specifications, displacement, and any other

particulars of the specific instruction type would be determined in accordance with the S/390 computer architecture. In this embodiment, this is done in the preprocessor and in the patching instructions, as necessary to provide their assigned functions.

FIG. 4 illustrates the layout of target storage which is used by the described embodiment. The incompatible processor emulator (SPE) area 402 (which manages exceptions) is shown in low storage. Part of the emulator storage SPE area 402 contains all the target processor translation template routines 407 for the incompatible instructions.

Each template routine is found through the Microcode Address Table, described earlier, in area 405. An area 406 is formatted to represent incompatible facilities that must be emulated, e.g. to emulate S/390, the PSW, the Control Registers and Access Registers would be here. This area also holds the target machine address translation tables and general emulator routines for initialization, dumping storage, etc. An area 401 is shown that is assigned for the incompatible real storage, containing incompatible instructions and data as they are allocated space in the incompatible environment by the incompatible operating system, middleware programming and the application programs. Incompatible system page faults are reported to and handled by the incompatible environment, which manages its allocation of storage as it would its own native storage in a non-emulated situation. If a target page fault is caused by an incompatible page being invalid (not backed by incompatible real storage), the incompatible page fault architecture for the page fault is emulated so the incompatible operating system can handle its page fault totally within the incompatible environment. Of course, this incompatible program execution is by emulation as described herein.

For incompatible storage area 401, the address of incompatible storage real address zero is shown as offset from the target real address zero SPE 0. All incompatible real addresses must be offset during emulation by this amount to find the required incompatible real storage location. This area of target storage is assigned and dedicated to incompatible storage on a byte-by-byte basis, one-for-one. The top part of the target storage map illustrates a very large target virtual address range 403 required to represent the access authority states of the incompatible machine architecture. U.S. Pat. No. 5,577,231 describes this in more detail.

FIG. 5 illustrates the process of accessing target instructions and patching them with information specified in the incompatible instruction being emulated. Each target processor translation template routine is written with complete knowledge of the incompatible machine instruction whose function it will perform, and it is coded to make the necessary modifications to the appropriate target instructions in the routine before executing those target instructions one at a time. Three special emulator mode instructions are used to provide the patch function and these are described here. These instructions provide general bit manipulation functions and, therefore, can be added to the microprocessor architecture and implemented as part of the processor chip.

Alternatively, they can be provided as microcoded instructions, or even as subroutines within the emulator, callable to provide the required patching function. These patch instructions are coded in the target translation template routine, as required, to perform the necessary modification of the functional instructions within the template routine. All target instructions are loaded to the Current Instruction Register 501 where they are executed by the processor.

Instructions requiring no modification before execution go directly from the template to 501. However, if a target instruction must be modified by incompatible instruction particulars before execution, it is first loaded to the Patch register 504 where all modifications are made before it is moved to 501 for execution. These actions are performed by three special Patch instructions provided in the processor emulation mode.

The LOAD FOR PATCH instruction depicted in FIG. 6 fetches the target emulation routine instruction immediately following the 'Load For Patch' instruction and loads it into the Patch Register 504.

The PATCH instruction depicted in FIG. 8 updates the Patch Register 504 with bits from the Incompatible Instruction Register 505, as specified by the operands of this instruction. The target bit locator operand 810 in FIG. 8 specifies the beginning bit position of the field in the Patch Register 504 which will be patched. The incompatible bit locator operand 811 specifies the beginning bit position of the patch data contained within the Incompatible Instruction Register 505. The length operand 812 in FIG. 8 specifies how many bits are moved.

The Patch instruction is used when more than one modification must be made to the target instruction before it is executed. The target instruction will not be executed until a Patch and Execute instruction is next executed in the target emulation routine. If only one modification to a target instruction is required before execution, Patch will not be used since Patch and Execute can perform the complete function.

The PATCH AND EXECUTE instruction depicted in FIG. 7 updates the Patch Register 504 with bits from the Incompatible Instruction Register 505, as specified by the operands of this instruction. The target bit locator operand 710 of FIG. 7 specifies the beginning bit position of the target instruction text in the Patch Register 504 which will be patched. The incompatible bit locator operand 711 in FIG. 7 specifies the beginning bit position of the patch data contained within the Incompatible Instruction Register. The length operand 712 in FIG. 7 specifies how many bits are moved. After the patch operation is completed, the contents of the Patch Register 504 are moved to the Current Instruction Register 501 to complete the execution of the Patch and Execute instruction. The patched target instruction is executed in the current instruction register 501.

The patch instructions will extract, from the patch information, the fields that will be used to modify target instructions in the target code template. Incompatible Instruction register 505 is the source of the patching information in all patching operations. The hardware patch register 504 is where the target instruction modification takes place. The first target instruction is addressed by the target instruction address register 212 as a consequence of the branch instruction executed by the preprocessor function at the end of its processing of an incompatible instruction. The instructions in the translation routine are executed normally by the processor by being loaded into the current instruction register 501 and then executed from there. This is illustrated in FIG. 5 by the lines 533, 538, and 539. Line 534 illustrates the path of a target instruction that must be patched before it can be executed. It is loaded to the patch register 504 where it is the object of the Patch and the Patch and Execute instructions, after which it is itself executed in its modified form. Thus, in terms of their order of execution by the target processor, the instructions of the translation routine which are shown in FIG. 5 are executed in the order 521, 523, 524, 522, 525.

To review, the patch instructions are Load for Patch, Patch, and Patch and Execute. Load for Patch places the

target instruction following the Load for Patch instruction in the template routine into the Patch Register 504. The Patch instruction extracts fields from the patch information, the incompatible instruction in this embodiment, and replaces fields in the target instruction in the Patch Register 504 with those extracted fields, to reflect the incompatible instruction instance as it was coded. Patch is useful where more than one patch operation is required on a target instruction to emulate the incompatible instruction. Patch and Execute performs the last extract and modify operation and then sends the resulting target instruction to the normal processor instruction register 501 for execution.

In the example shown, in incompatible instruction register 505 two fields from the incompatible instruction modify the target instruction before it is executed. Each instruction is loaded into the processor current instruction register 501 where it is executed by normal processor functions. Box 520 contains the instructions of the template of the example, labeled 521, 522, 523, 524 and 525. However, target instructions that must be modified before execution first are loaded into the Patch Register 504, where all modifications are made, and then loaded to the Current Instruction Register 504 for execution.

The Load for Patch instruction loads the next instruction 522 into the Patch Register 504. This is illustrated in FIG. 5 by line 534. The next instruction executed by the target processor in emulator mode is the Patch instruction 523 in the template routine. Its action is depicted by line 535 in FIG. 5. It moves the indicated field from the incompatible instruction image in 505 to replace the indicated field in the target instruction in the Patch Register 504. The next instruction executed is the Patch and Execute 524 in the template routine. Its first action is depicted by line 536, replacing a second field in the target instruction with a different field from the incompatible instruction.

After the patch operation is complete, in the Patch and Execute instruction's second action, depicted by line 537, the target instruction is presented to the processor for normal execution in its modified state by being moved into instruction register 501. Thus, the instructions in the template are actually executed by the processor in the order 521, 523, 524, 522, 525. Target instruction 525 signals the end of the template and directs the processor back to the preprocessor to decode the next incompatible instruction.

In this example, in emulating S/390 on a PowerPC RISC processor (the target processor), the S/390 LOAD REGISTER instruction is emulated by a PowerPC ADD IMMEDIATE instruction. The S/390 LOAD REGISTER, shown as the content of the incompatible instruction register 505 in FIG. 5, is two bytes in length. Bits 0–7 contain the operation code ('18'x), bits 8–11 contain the R1 register, or target, and bits 12–15 contain the R2 register. In the example presented in FIG. 5, R1=2, and R2=3. The PowerPC ADD IMMEDIATE instruction, shown as the target instruction loaded into the patch register by the Load for Patch instruction in FIG. 5, is four bytes in length. Bits 0–5 contain the operation code ('14'x), bits 6–10 contain the RT register, or target, bits 11–15 contain the RA register, or incompatible instruction, and bits 16–31 contain the SI field, or immediate data. When emulating a LOAD REGISTER instruction, the SI field of the PowerPC ADD IMMEDIATE instruction is zero. In FIG. 5, LOAD FOR PATCH loads the Patch Register 504 with the ADD IMMEDIATE skeleton. A Patch instruction is used to move 4 bits beginning at bit position 8 of the Incompatible Instruction Register 505 to the Patch Register 504 beginning at bit position 6. A Patch And Execute instruction is used to move 4 bits beginning at bit position 12 of the Incompatible Instruction Register to the Patch Register beginning at bit position 11, and then load the Current Instruction Register 501 with the fully patched ADD IMMEDIATE instruction for execution by the processor.

Completion of a target processor translation template routine is signaled by an End Routine instruction found at the end of each target translation template routine. This directs the processor to return to the preprocessor function to decode the next incompatible instruction and to find the template routine address for the next incompatible instruction. A template routine may contain many sets of these patch instructions, including multiple Patch and Execute instructions for multiple target instructions of a single incompatible instruction target translation template routine. Also, target instructions will be found in the translation templates that require no patching and will execute in register 501, after being moved there directly, as the patch instructions themselves execute.

FIGS. 9 through 15 show the process of preprocessing, patching, and executing instructions in emulation mode. FIG. 9 depicts the operation of the preprocessing function. Step 901 accesses the content of the incompatible instruction address register to find out the location of the next incompatible instruction to be executed. It uses this address to access the incompatible instruction in step 902, and loads it into the incompatible instruction register 505 in step 903. At step 904, it decodes the incompatible instruction in accordance with incompatible processor instruction format architecture to obtain its opcode and length. The length is added to the incompatible instruction address register 204 at step 905. Step 906 examines certain selected fields of the instruction for null. These fields are chosen because experience has shown that they are often coded as null, and if they are null, the target translation routine assuming them to be null can contain fewer target instructions and therefore provide faster emulation of those instruction instances.

A bit is assigned for each field that is examined for null in the incompatible instructions, with one state meaning null and the other meaning that the field was coded non-null in the instance being examined. The field of these bits is appended to the incompatible instruction opcode in step 907 to form the value used for a table lookup in step 908 in the Microcode Address Table 205 to find the address of the proper target translation template routine to perform the function of the incompatible instruction. Step 909 branches to the first instruction of the selected translation routine, completing preprocessing on the current incompatible instruction. Control passes to entry point 10A on FIG. 10 which depicts execution of the instructions in the translation routine.

In FIG. 10 at step 1001 the instruction addressed by the instruction address register 212 is loaded into the processor's current target instruction register 501 for execution. This is depicted in FIG. 5 by the line labeled 533. Box 1001 tests the target processor operation code in the current instruction register to ascertain if it is one of the patch instructions, or the End Routine instruction which indicates the end of the translation template routine. If the instruction is End Routine, Load for Patch, Patch, Patch and Execute, Ifetch Reset or Set Authority, control passes to the appropriate process starting at entry points 9A, 11A, 12A, 13A, 14A or 15A. If not, at step 1008, the processor executes the instruction. The target instruction address register 212 is incremented at step 1009, and control returns to step 1001 to interpret the next target instruction.

Step 1002 tests the instruction in the current instruction register to ascertain whether or not it is an End Routine. If it is, control passes to entry point 9A on FIG. 9. If not, the instruction is tested at step 1003 for being a Load for Patch instruction. If it is, control passes to entry point 11A on FIG. 11. If not, the instruction is tested for being a Patch instruction at step 1004. If it is, control passes to entry point 12A on FIG. 12. If not, the instruction is tested for being a Patch and Execute instruction at step 1005. If it is, control passes to entry point 13A on FIG. 13. If not, the instruction is tested for being an Ifetch Reset. If it is, control passes to entry point 14A on FIG. 14. If not, the instruction is tested for being a Set Authority. If it is, control passes to entry point 15A on FIG. 15. If not, the instruction is executed at 1008, the instruction count in register 501 is incremented at step 1009 and control returns to entry point 10A at 1001. FIG. 11 describes the processing of the Load for Patch instruction which, at step 1101, increments the target instruction count in register 212 to address the instruction following the Load for Patch instruction, which following instruction is to be loaded into the patch register.

Then step 1102 loads the patch register 504 with that next instruction in the template routine, which is now addressed by the instruction count in register 212. Control returns to entry point 10B on FIG. 10 to process the next instruction in the translation template. FIG. 12 describes the processing of the Patch instruction, which moves particulars of the incompatible instruction image to overwrite portions of one of the target processor instructions that will perform the equivalent function. At step 1201, the field, starting at an offset specified in the Patch instruction and of a length specified in the Patch instruction, as shown in FIG. 8, is accessed in the incompatible instruction register 505 and moved to overwrite the instruction in the Patch register 504 starting at the offset specified in the Patch instruction for that purpose. This is illustrated in FIG. 5 by the line labeled 535. Control passes to entry point 10B on FIG. 10.

FIG. 13 shows the processing of the Patch and Execute instruction. The incompatible instruction field specified in the instruction is accessed in the incompatible instruction register 505 and overwrites the specified field specified in the instruction, whose format is shown in FIG. 7, in the patch register 504, at step 1301. This is illustrated by the line labeled 536 in FIG. 5. The content of the patch register 504 is then loaded to the current instruction register 501 in step 1302. This is illustrated in FIG. 5 by the line labeled 537. Control passes to entry point 10C on FIG. 10, in order to execute the modified instruction.

FIG. 14 shows the processing of the Ifetch Reset instruction. Step 1401 loads the incompatible instruction address register 204 with the address specified by the Ifetch Reset instruction operand. Control returns to step 9A on FIG. 9 to preprocess the incompatible instruction at the new incompatible execution location.

FIG. 15 shows the processing of the Set Authority instruction. At step 1501 the access authority state value in 203 and 305 is replaced by the new value specified by the Set Authority instruction operand 1801. Control returns to step 10B on FIG. 10.

FIGS. 16, 17, and 18 show the format of the End Routine, the IFETCH Reset and the Set Authority instructions.

FIG. 19 depicts the instruction translation template for execution on a PowerPC processor of a S/390 load-register-from-storage instruction, whose format is shown in the Figure. Here an effective address must be used to access incompatible storage area, calculated in incompatible architecture by performing the sum of the content of the specified B register and the content of the specified X register plus the displacement D. R specifies the incompatible register to be loaded from incompatible storage. The emulation of an S/390 LOAD instruction, specifying both a B register and an X register, requires three PowerPC instructions. First an ADD instruction is needed to add the contents of the B and X registers. Then an RLDIMI instruction is needed to insert the resulting address value into the address portion of the DEVAR, thereby providing the incompatible data address with the needed authorization state attached. Finally, an LWZ instruction loads the appropriate R register from the specified address with displacement D added. The target instructions need to be patched to contain the S/390 particulars, R, X, B and D. The template in FIG. 19 begins with a Load For Patch, specifying the ADD instruction. The Patch instruction is used to update the ADD instruction with the S/390 X value. The Patch and Execute instruction updates the ADD instruction with the S/390 B value and then completes by executing the ADD instruction which combines the contents of the B and X registers. The next target instruction executed does not require patching. The PowerPC RLDIMI instruction causes the address calculated by the ADD instruction to be inserted into the address portion of the DEVAR register. The DEVAR register contains the S/390 authority state in the high-order bit positions.

The template next contains a Load For Patch instruction specifying the PowerPC LWZ instruction. The following Patch instruction updates the LWZ to contain the S/390 D value. Next, the Patch and Execute instruction updates the LWZ instruction with the S/390 R value and completes the execution of the LWZ instruction, which loads R from the incompatible data area specified by the sum of the S/390 B, X and D values. The template routine ends with the End Routine instruction, which transfers target processor execution to the preprocessor to start execution of the next incompatible incompatible instruction.

FIG. 20 illustrates a different template for the same S/390 Load-register-from-storage instruction type, where the preprocessor finds that no X register field has been specified. A separate template routine is provided for such a case. Since the X field of the incompatible instruction instance to be executed is zero, only the address in the specified B register, as modified by the addition of the displacement D, supplies the incompatible storage address. Since X is not specified, the target processor ADD instruction to combine the contents of the B and X registers is not needed, allowing a shorter template to be specified. As a result, execution of the incompatible instruction is faster than if the template in FIG. 19 were used for all cases of that incompatible instruction type. This example is used to illustrate the flexibility of the invention in providing a tradeoff between the complexity of the preprocessor in decoding incompatible instructions, and the performance obtained from the emulation.

In the case of S/390, several cases of fields often not coded in certain classes of instructions can be determined. A tradeoff can be made as to the performance benefit of having these cases recognized by the preprocessor on a one-for-one basis, with separate templates provided for their execution. Examples of fields that can be checked for null values are displacements, X register fields, B register fields, with corresponding simpler target instruction template routines where those fields are null in the incompatible instruction instance to be executed.

The six special instructions have been described in the embodiment as instructions. As such they can be implemented as hardware functions of a microprocessor or as

microcoded instructions. Alternatively, they can be expressed as macroinstructions and compiled as program calls to routines in the emulator that perform the required functions when called. Also, although the preferred embodiment shows the IEVAR 203 and the DEVAR 305 as separate registers, other embodiments may use the same target hardware register to perform both functions, and it may be termed the incompatible-instruction data address register since the target process accesses the incompatible instructions as its data. Other embodiments may use several target registers to hold the addresses of the incompatible instructions and the addresses of the operand data accessed by the incompatible instructions.

While we have described our preferred embodiments of our invention, it will be understood that those skilled in the art, both now and in the future, may make various improvements and enhancements which fall within the scope of the claims, which follow. These claims should be construed to maintain the proper protection for the invention first disclosed herein.

What is claimed is:

1. An emulation method for executing an incompatible computer program on a target processor, the incompatible program containing computer instructions natively executable on a different processor built to a computer architecture incompatible with target architecture used for building a target system containing the target processor, the target architecture defining target instructions executable on the target processor and the incompatible architecture defining execution results for the incompatible instructions, the emulation method comprising

storing in a target system memory one or more target routines for each operation code provided in the incompatible instructions of the incompatible program, and each target routine performing a function similar to, but not necessarily identically to, a corresponding incompatible instruction,

associating one or more patching instructions with a target instruction in a target routine when the target instruction requires modification for enabling the target routine to provide execution results identically to execution results defined for the corresponding incompatible instruction by the incompatible architecture, and not associating any patching instruction with any target instruction not requiring modification for enabling the target routine to provide execution results identically to the corresponding incompatible instruction, each target instruction in a target routine being associated with none, one, or more patching instruction(s) of the target routine,

accessing in an emulation sequence the incompatible instructions stored as data in a target system memory, the emulation sequence being determined by sequentially selecting each next incompatible instruction to be next emulated for the incompatible program except when emulation results obtained for a previously emulated incompatible instruction determines the next incompatible instruction to be non-sequential during the emulation of the incompatible program,

utilizing contents of each accessed incompatible instruction, including at least the opcode, to select the corresponding target routine for emulating the accessed incompatible instruction,

preprocessing each target instruction in the corresponding target routine having one or more associated patching instructions for modifying the target instruction for enabling execution of the target routine to provide execution results identical to execution results defined for the accessed incompatible instruction, and no preprocessing being done for any target instruction not having any associated patching instruction, and

executing by the target processor each target instruction in the corresponding target routine after any preprocessing is completed for modifying the target instruction, and executing each target instruction without preprocessing if no patching instructions are associated with the target instruction.

2. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 1, the emulation method further comprising

accessing a next preprocessing instruction in the corresponding target routine after an execute type of patching instruction is detected to indicate modification of a last preprocessed target instruction is completed by execution of one or more patching instructions.

3. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 2, the emulation method further comprising

accessing the next incompatible instruction for the incompatible program after the next preprocessing instruction in the corresponding target routine is detected to be an end routine type of preprocessing instruction indicating completion of execution of the corresponding target routine.

4. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 3, the emulation method further comprising

execution of the one or more target instructions in the corresponding target routine performing accesses of data operands of the corresponding incompatible instruction in the target system memory, and utilizing the accessed data operands for generating execution results of the incompatible instruction.

5. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 4, the emulation method further comprising

determining a data address for a next incompatible instruction to be accessed in the target system memory by adding the length of a last accessed incompatible instruction to the address in the target system memory of the last accessed incompatible instruction, and resetting the data address for the next incompatible instruction to target processor execution results of the last accessed incompatible instruction when the execution results indicate a branch in the execution sequence of the incompatible instructions for the incompatible program.

6. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 5, the emulation method further comprising

concatenating an authority value with the data address to form an exploded virtual address for accessing a next incompatible instruction in the target system memory, and the authority value being provided by target processor execution results of one or more previously emulated incompatible instructions.

7. An emulation method for executing an incompatible computer program on a target processor, the incompatible program containing computer instructions executable on a different processor built to a computer architecture incompatible with target architecture used for building a target system containing the target processor, the target architec-

ture defining target instructions executable on the target processor and the incompatible architecture defining execution results for the incompatible instructions, the emulation method comprising

storing in a target system memory of the target processor at least one target routine for each operation code found in the incompatible instructions of the incompatible program, each target routine including one or more target instruction(s), and each incompatible instruction in the incompatible program associated with a corresponding target routine accessed by utilizing information obtained in a corresponding incompatible instruction,

associating one or more patching instructions with each target instruction requiring preprocessing modification for enabling the target routine to function identically to the corresponding incompatible instruction, and not associating any patching instruction with any target instruction not requiring preprocessing modification for enabling the target routine to function identically to the corresponding incompatible instruction,

wherein each target instruction in a target routine may be respectively associated with none, one, or more than one, patching instruction(s),

executing a target instruction after completing any preprocessing modification if one or more patching instructions are associated with the target instruction, and executing a target instruction without preprocessing modification if no patching instructions are associated with the target instruction, and executing the target instructions in a sequence indicated in the corresponding target routines, and

completing emulation of each corresponding incompatible instruction in the incompatible program when all target instructions have been executed in the corresponding target routine.

8. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 7, the emulation method further comprising

storing in the target system memory a plurality of target routines for one operation code used in the incompatible instructions of the incompatible program, the plural target routines being associated with different forms of incompatible instruction instances using the one operation code, the different forms including specifying a null value or a non-null value for a like operand of the incompatible instruction instances using the same operation code, wherein a lesser number of patching instructions are in the target routine for the incompatible instruction instances having a null-value operand than in a target routine for the incompatible instruction supporting all operands as having any values including both null and non-null values, wherein target system performance is improved when the corresponding target routine emulates an incompatible instruction instance having at least one null operand by enabling the corresponding target routine to execute a smaller number of target instructions.

9. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 7, the emulation method further comprising

detecting a load-for-patch instruction in a target routine for identifying an associated target instruction as requiring modification by a patch operation prior to being in a condition for execution by the target processor, and

detecting a target instruction not associated with any load-for-patch instruction as a target instruction which is in a condition for execution by the target processor.

10. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 9, the emulation method further comprising

locating one or more patching instruction(s) associated with the target instruction for specifying one or more patching operation(s) on the target instruction, each patching instruction defining one or more field(s) of bits in the target instruction to be modified by one or more field(s) of bits defined in the corresponding incompatible instruction.

11. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 10, the emulation method further comprising

executing an execute-type of preprocessing instruction in the target routine to indicate patching modification is completed for an associated target instruction.

12. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 11, the emulation method further comprising

executing the associated target instruction in response to execution of the execute-type of preprocessing instruction which indicates all modification of the associated target instruction is completed.

13. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 11, the emulation method further comprising

executing on the target processor a patching operation on the associated target instruction as part of the execution of the execute-type of preprocessing instruction before executing the associated target instruction.

14. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 13, the emulation method further comprising

associating with a target instruction the load-for-patch instruction and additional patching instruction(s) by locating the these instructions at predetermined location(s) relative to the target instruction in the associated target routine, and locating an execute-type of preprocessing instruction as a last patching instruction associated with the target instruction to initiate complete modification and to initiate execution of the associated target instruction by the target processor.

15. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 7, the emulation method further comprising

storing in the target system memory at least one target routine for each operation code defined in the incompatible architecture, to enable the target processor to emulate any incompatible program based on the incompatible architecture.

16. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 15, the emulation method further comprising

fetching each incompatible instruction as data from the target system memory for representing the next incompatible instruction to be emulated in an incompatible program,

interpreting each fetched incompatible instruction to determine its operation code, instruction type, instruction format, operand locations, and patch field information, and

utilizing at least the operation code to access an entry in a coded routine-location table in the target system memory, the entry containing a target address of a corresponding target routine in the memory for emulating the incompatible instruction.

17. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 16, further comprising

storing a target routine address obtained from the table for the corresponding target routine to make the target routine address available to the target processor, and

also storing with the target routine address other instruction information obtained from the incompatible instruction.

18. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 7, the emulation method further comprising

obtaining by the target processor each next target routine address in the order that target routine addresses are obtained from the routine address table for an emulation sequence of incompatible instructions,

utilizing each next target routine address for accessing a next target routine for execution by the target processor, and

preprocessing any patching instructions for each target instruction in the target routine for modifying the target instruction, and

executing all target instructions in execution sequence in the target routines whether or not any target instruction is associated with any patching instruction(s).

19. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 18, the emulation method further comprising

accessing any next target instruction in a corresponding target routine,

executing by the target processor any patching instructions associated with the target instruction, then executing the target instruction if any required patching is completed, and executing the target instruction without any patching modification if no patching instructions are associated with the target instruction, and

repeating the accessing and executing steps for each next target instruction to be executed in the corresponding target routine until all target instructions in the corresponding target routine are executed.

20. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 7, the emulation method further comprising

fetching each incompatible instruction from the target system memory as a next incompatible instruction to be emulated in the incompatible program,

interpreting an operation code of each fetched incompatible instruction to determine type, format and patch field information of the incompatible instruction,

utilizing at least the operation code to access an entry in a coded table in target system memory, the entry containing a target routine address of a corresponding target routine in the memory for emulating the incompatible instruction,

determining an instruction length for the incompatible instruction, and providing the instruction length to an instruction address register for generating an address therein of the next sequential incompatible instruction in the incompatible program, and

storing the address of the next sequential incompatible instruction provided by the instruction address register in an incompatible instruction address register for accessing a next incompatible instruction from the target system memory.

21. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 20, the emulation method further comprising

utilizing the target routine address obtained by the target processor for accessing a first target instruction in a corresponding target routine, and

executing by the target processor any patching instruction (s) associated with the first target instruction, and then executing the target instruction after any patching instruction execution is completed, and executing the first instruction without executing any patching instruction if no patching instruction is associated with the target instruction.

22. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 21, the emulation method further comprising

accessing by the target processor each next target instruction in the corresponding target routine if any target instruction(s) follow the first target instruction in the target routine,

executing by the target processor any patching instructions associated with each next target instruction, and then executing the target instruction after any required patching is completed, but executing the target instruction without executing any patching instruction if no patching instruction is associated with the target instruction, and

repeating the accessing and executing steps for each next target instruction in the corresponding target routine until all target instructions in the corresponding target routine are executed.

23. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 22, the emulation method further comprising

storing an image of the computer instruction in addition to the target routine address and incompatible instruction address, to represent fields in the incompatible instruction to be used by patching instructions for patching one or more target instructions in a corresponding target routine.

24. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 23, the emulation method further comprising

determining an instruction length for the incompatible instruction, adding the instruction length to a current instruction address in an incompatible instruction address counter for generating an address of a next sequential incompatible instruction in the incompatible program,

storing the generated address in the incompatible instruction address counter as a next incompatible instruction address for accessing a next incompatible instruction as target instruction data from the target system memory, and

providing the next incompatible instruction address to an incompatible instruction address register for accessing the next incompatible instruction as target data from the target system memory.

25. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 24, the emulation method further comprising

setting a low-order section of the incompatible instruction data address register to the next incompatible instruction address provided by an incompatible instruction address counter,

setting a high-order section of the incompatible instruction data address register to an access authority state, and

utilizing the entire content of the incompatible instruction data address register as a virtual address in the target system memory in which the high-order section supports incompatible authority controls of the incompatible architecture in the target system, even though the incompatible authority controls are transparent to the target processor.

26. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **25**, the emulation method further comprising

branching in the emulated execution of the incompatible program by replacing the low-order section of the incompatible instruction data address register with a branch address obtained by the target processor from execution of a corresponding target routine emulating the execution of an incompatible branching instruction.

27. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **26**, the emulation method further comprising

controlling access authority within the emulated execution of the incompatible program by replacing the high order section of the incompatible instruction data address register with an authority value obtained by the target processor from execution of a corresponding target routine emulating the execution of an incompatible authority control instruction.

28. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **27**, the emulation method further comprising

sizing the incompatible instruction data address register to contain a number of bits at least equal to the number of bits in each target processor virtual address, and

separating the content of the incompatible instruction data address register into an incompatible instruction address section and an authority section, in which the size of the incompatible instruction address section has at least a number of bit positions equal to the size of each incompatible program address, and the size of the authority section is up to the remaining bit positions in the target processor virtual address not used for containing the incompatible instruction address.

29. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **28**, the emulation method further comprising

mapping authority areas in the target processor's virtual memory, an authority area being located in the target processor's virtual memory by a target virtual address comprised of the authority value concatenated with zero-value low-order bits equal in number to the size of the incompatible instruction address.

30. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **9**, the emulation method further comprising

loading a target instruction following each load-for-patch instruction into a register of the target processor, and

modifying the target instruction in the register as specified by each following patching instructions located prior to any following load-for-patch instruction.

31. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **29**, the emulation method further comprising

executing a next-incompatible-instruction routine in the target processor for obtaining a next incompatible instruction address for the incompatible instruction counter to be used for fetching data from the target system memory for representing a next incompatible instruction by executing a target instruction in a target routine to obtain a target instruction operand which is the required data representing the next incompatible instruction in the target system, since the next incompatible instruction is stored as data in the target system memory, wherein processing for the incompatible instructions by the target processor is never recognized by the target processor as execution of the incompatible instructions.

32. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **31**, the emulation method further comprising

initiating the execution of the next-incompatible-instruction target routine for fetching data representing a next incompatible instruction upon completion of execution for each target routine for an incompatible instruction.

33. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **32**, the emulation method executing a next-incompatible-instruction target routine for:

fetching from the target system memory operand data of a target instruction representing a next incompatible instruction upon completion of execution by the target processor of each target routine for an incompatible instruction, and

interpreting content of each fetched incompatible instruction to determine fields in the incompatible instruction to be used for selecting an address of a corresponding target routine from a table of target routine addresses in the target system memory.

34. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **33**, the emulation method further comprising

executing a target instruction for utilizing at least the operation code interpreted in the fetched incompatible instruction to select from the table of target routine addresses an address of a corresponding target routine,

putting in a target processor register the address of the corresponding target routine selected from the table, and optionally putting in the register the address of the incompatible instruction and an image of the incompatible instruction to represent fields available to patching instructions for modifying one or more target instructions in the corresponding target routine, and

accessing the target routine in target system memory for execution by the target processor.

35. An emulation method for executing an incompatible computer program on a target processor, as defined in claim **34**, the target routine interpreting the fetched incompatible-instruction further performing the steps of:

generating a next address for accessing a next sequential incompatible instruction in the incompatible program by adding an instruction length of the fetched incompatible instruction to a current incompatible-instruction-address counter for generating the next address which will be used for sequentially fetching a next incompatible instruction of the incompatible program, and

storing the generated next address in an incompatible instruction address register for fetching the next incompatible instruction in the target system memory.

36. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 35, the last storing step in the emulation method further comprising

setting a low-order section of the incompatible instruction data address register to the generated next operand address,

setting a high-order section of the incompatible instruction data address register to an access authority state, and

utilizing the entire content of the incompatible instruction data address register as a virtual address in the target processor memory for enforcing authority controls of the incompatible architecture in the target system.

37. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 36, the emulation method further comprising

branching in the emulated execution of the incompatible program by replacing the low-order section of the incompatible instruction data address register with a branch address provided by the target processor executing a target routine emulating the execution of an incompatible branching instruction.

38. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 37, the emulation method further comprising

resetting the high-order section of the incompatible instruction data address register by the target processor executing a target routine providing an access authority value while emulating an authority-control incompatible instruction.

39. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 38, the emulation method further comprising

checking the access authority value in the high-order section of the incompatible-instruction data address register during execution of an incompatible program to determine if the access authority value is the value required to make the storage access to the incompatible instruction.

40. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 39, the emulation method further comprising

setting a low-order section of a target-processor register used as an incompatible-instruction data address register for containing an incompatible-instruction effective address for a storage operand of the incompatible instruction being emulated by executing one or more target instructions for the target routine to compute the incompatible-instruction effective data address and put it into the target-processor register,

setting a high-order section of the target incompatible-instruction data address register to an access authority assigned to the incompatible program being emulated,

utilizing the entire content of the target incompatible-instruction data address register as a target system virtual address for accessing the operand data in a target system memory, and

translating the target system virtual address to locate the operand data in the target system memory from which the operand data is copied to the target processor for use by the target routine in completing execution of the incompatible instruction, in which the high-order section supports incompatible authority controls of the incompatible architecture in the target system even though the incompatible authority controls are transparent to the target processor.

41. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 40, the emulation method further comprising

again setting the low-order section of the incompatible-instruction data address register to another incompatible-instruction effective data address for another storage operand of the incompatible instruction being emulated by executing the one or more target instructions for the target routine to compute the another incompatible-instruction effective data address,

utilizing the entire content of the target incompatible-instruction data address register as a target system virtual address for accessing the another operand data in the target system memory without changing the high-order section if a determination of the access authority for the another operand data finds the same access authority exists for the another operand data of the incompatible program being emulated, and

translating the target system virtual address to locate another operand data in the target system memory from which the another operand data is copied to the target processor for use by the target routine in completing execution of the incompatible instruction when the incompatible instruction has more than one storage operand.

42. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 41, the emulation method further comprising

sizing the target-processor register used as an incompatible-instruction data address register as containing a number of bits at least equal to the number of bits in each target processor virtual address, and

determining an address section of the incompatible-instruction data address register as having a number of bit positions at least equal to the size of each target processor effective data address, and an access authority section having up to the remaining bit positions in the target processor virtual address not used by the address section.

43. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 38, the emulation method further comprising

checking the access authority value in the high-order section of the incompatible-instruction data address register during execution of an incompatible program to determine if the access authority value is the value required to make the data access to the requested incompatible operand data.

44. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 7, the preprocessing in the emulation method further comprising

including preprocessing instructions in any target routine for emulating an incompatible instruction preprocessing instruction for which any preprocessing instruction is: 1) for modifying a target instruction in the target routine prior to emulation execution of the target instruction, or 2) for contolling a target system virtual address for accessing another incompatible instruction as data in the target system storage.

45. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 44, the preprocessing in the emulation method further comprising

executing each preprocessing instruction by microcoded software in the target system for use by each target processor in the system which is to perform preprocessing of target routines used for emulating the incompatible instructions in an incompatible program, and

protecting the microcoded software from accessing by operating-system software or by application software executing on any processor in the target system.

46. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 44, the preprocessing in the emulation method further comprising

executing each preprocessing instruction by a microcoded software routine for simulating the function of the preprocessing instruction,

structuring the macrocoded software routine from target processor instructions available in current target system architecture, and

locating the macrocoded software routine in the target system storage.

47. An emulation method for executing an incompatible computer program on a target processor, as defined in claim 44, the preprocessing in the emulation method further comprising

including as preprocessing instructions: any patching instruction in a target routine, and any address-changing instruction in a target routine for changing a data address to be used for accessing a next incompatible instruction, or for changing any address component in the data address whether the data address is a target real address, a target absolute address, or a target virtual address, and whether the address change is in an address component of the data address which is an authority authorization value, and address space identifier value, or an address value within an address space.

* * * * *

EXHIBIT 17

US005687106A

United States Patent [19]

Schwarz et al.

[11] **Patent Number: 5,687,106**

[45] **Date of Patent: Nov. 11, 1997**

[54] **IMPLEMENTATION OF BINARY FLOATING POINT USING HEXADECIMAL FLOATING POINT UNIT**

[75] Inventors: **Eric Mark Schwarz**, Gardiner; **Charles Franklin Webb**; **Kai-Ann Ho**, both of Poughkeepsie, all of N.Y.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **414,250**

[22] Filed: **Mar. 31, 1995**

[51] Int. Cl.[6] **G06F 7/38**; G06F 7/00

[52] U.S. Cl. **364/748**; 364/715.03

[58] Field of Search 364/715.01, 715.03, 364/745, 748; 341/89, 94, 104, 105

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,792,793	12/1988	Rawlinson et al.	341/89
4,942,547	7/1990	Joyce et al.	364/748
4,949,291	8/1990	Saini	364/715.03
5,058,048	10/1991	Gupta et al.	364/748
5,161,117	11/1992	Waggener, Jr.	364/715.03
5,191,335	3/1993	Leitherer	341/94
5,235,533	8/1993	Sweedle	364/715.03
5,249,149	9/1993	Cocanougher et al.	364/748
5,267,186	11/1993	Gupta et al.	364/748
5,305,248	4/1994	Ammann	364/748
5,307,301	4/1994	Sweedler	364/748
5,309,383	5/1994	Kuroiwa	364/748
5,337,265	8/1994	Desrosiers et al.	364/748
5,523,961	6/1996	Naini	364/715.03

FOREIGN PATENT DOCUMENTS

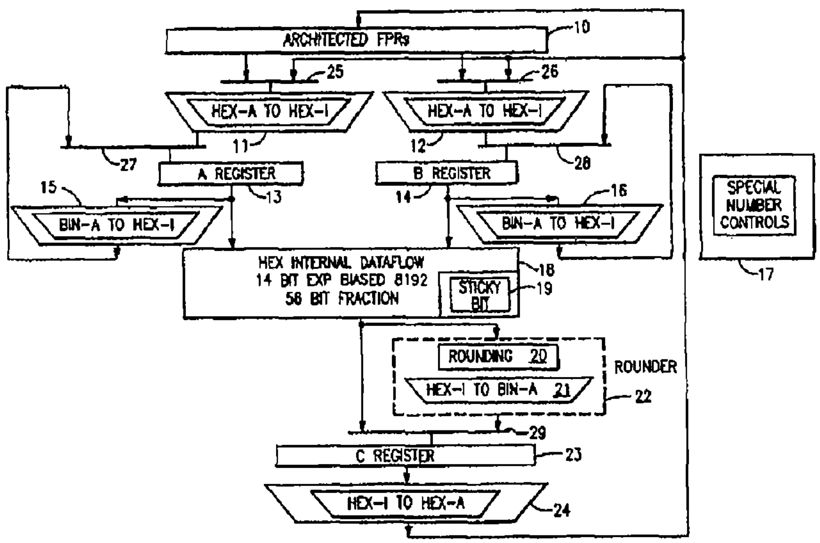
62-249230	10/1987	Japan .
1217622	8/1989	Japan .
1237822	9/1989	Japan .
1290535	2/1987	U.S.S.R. .
89/10189	7/1989	WIPO .

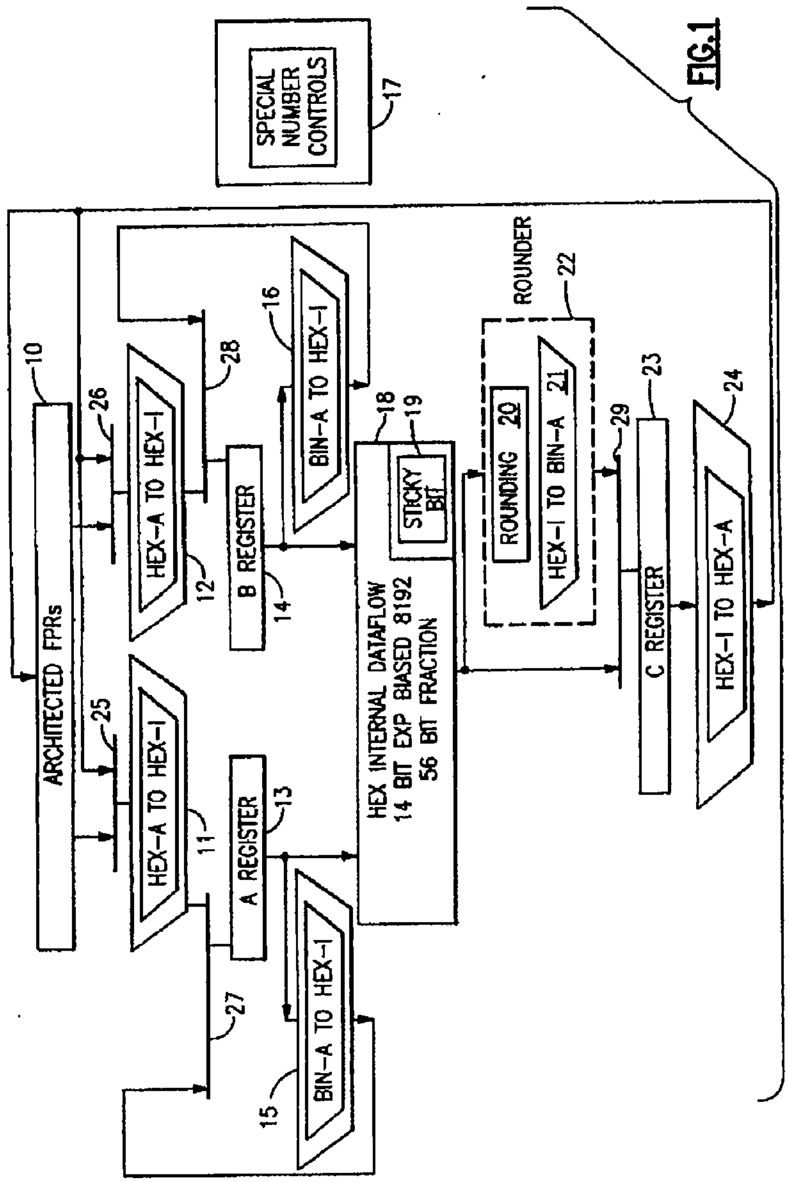
Primary Examiner—Chuong D. Ngo
Attorney, Agent, or Firm—Lynn L. Augspurger

[57] **ABSTRACT**

A computer system supporting multiple floating point architectures. In an embodiment of the invention, a floating point unit (FPU) is optimized for hex format. The FPU uses a hex internal dataflow with a with an exponent and bias sufficient to support a binary floating point architecture. The FPU includes format conversion means, rounding means, sticky bit calculation means, and special number control means to execute binary floating point operations according to the IEEE 754 standard. An embodiment of the invention provides a system for executing floating point operations in either IBM S/390 hexadecimal format or IEEE 754 binary format.

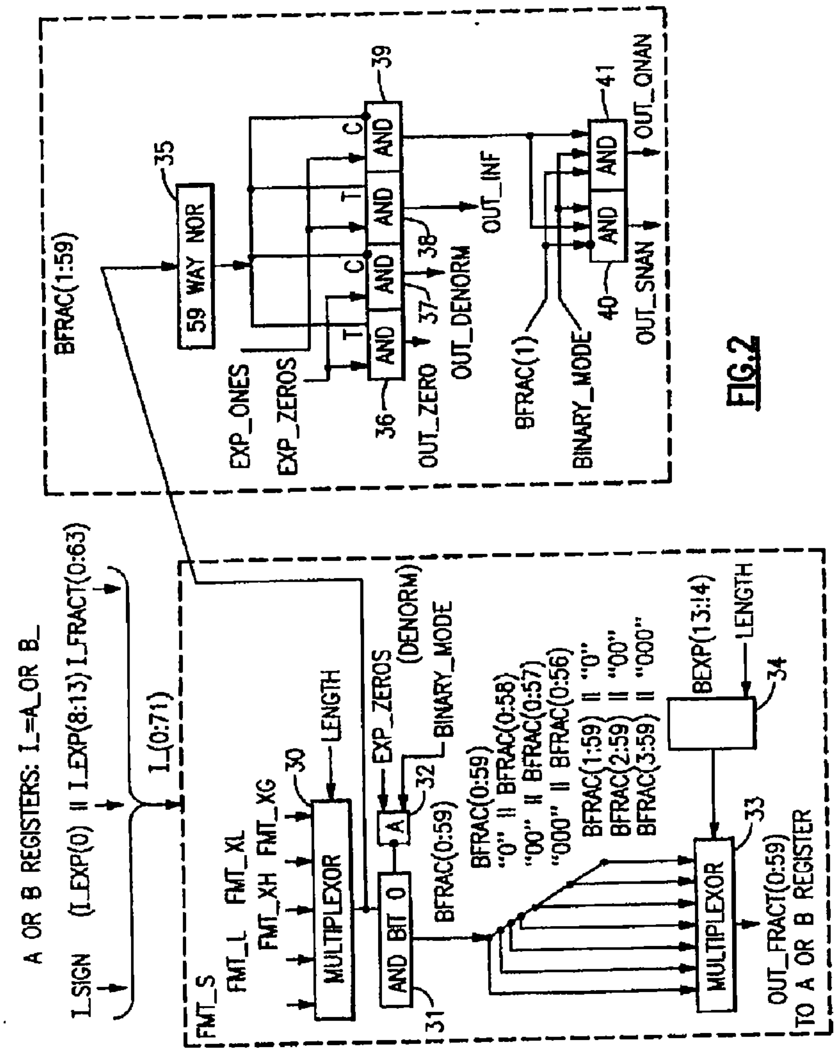
17 Claims, 5 Drawing Sheets





SPECIAL NUMBER CONTROLS
17
10
ARCHITECTED FPRs
26
25
HEX-A TO HEX-I
12
HEX-A TO HEX-I
11
28
27
B REGISTER
14
A REGISTER
13
16
BIN-A TO HEX-I
15
BIN-A TO HEX-I
18
19
STICKY BIT
HEX INTERNAL DATAFLOW
14 BIT EXP BIASED 8192
56 BIT FRACTION
ROUNDER
22
ROUNDING 20
HEX-I TO BIN-A 21
29
23
C REGISTER
24
HEX-I TO HEX-A

FIG.1



BFRAC(1:59)
59 WAY NOR
35
EXP_ONES
EXP_ZEROS
AND
AND
AND
AND
36
37
38
39
OUT_ZERO
OUT_DENORM
OUT_INF
BFRAC(1)
BINARY_MODE
AND
AND
40
41
OUT_SNAN
OUT_QNAN
A OR B REGISTERS: I_=A_OR B_
I_SIGN (I_EXP(0) II I_EXP(8:13) I_FRACT(0:63)
I_(0:71)
FMT_S
FMT_L FMT_XL
FMT_XH FMT_XG
MULTIPLEXOR
30
LENGTH
AND BIT 0
31
A
32
EXP_ZEROS
(DENORM)
BINARY_MODE
BFRAC(0:59)
BFRAC(0:59)
"0" II BFRAC(0:58)
"00" II BFRAC(0:57)
"000" II BFRAC(0:56)
BFRAC(1:59) II "0"
BFRAC(2:59) II "00"
BFRAC(3:59) II "000"
BEXP(13:14)
LENGTH
34
33
MULTIPLEXOR
OUT_FRACT(0:59)
TO A OR B REGISTER

FIG.2

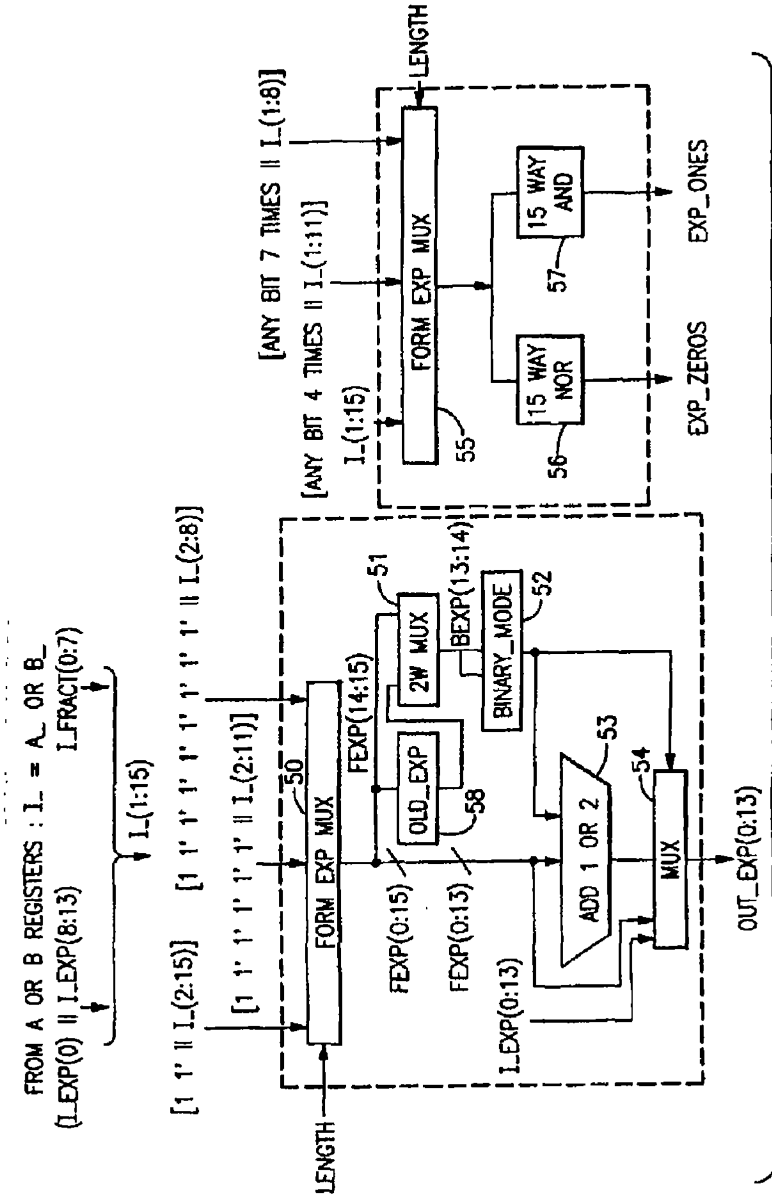
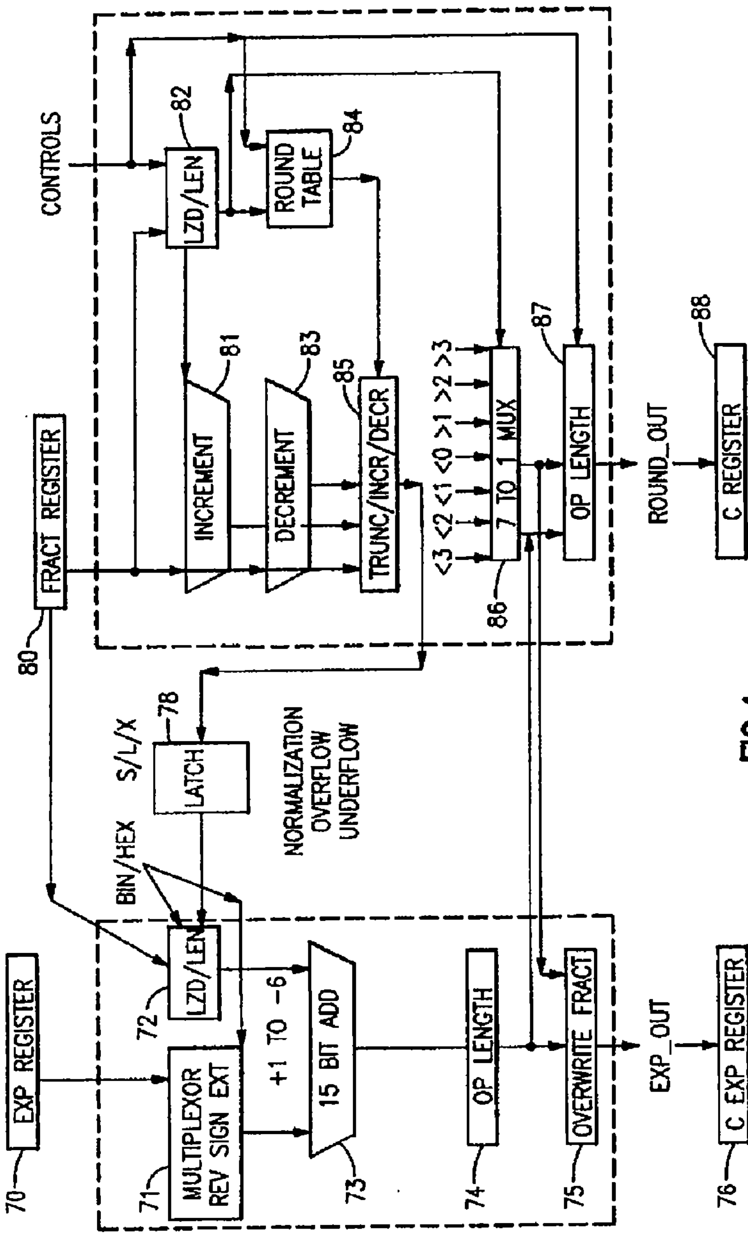
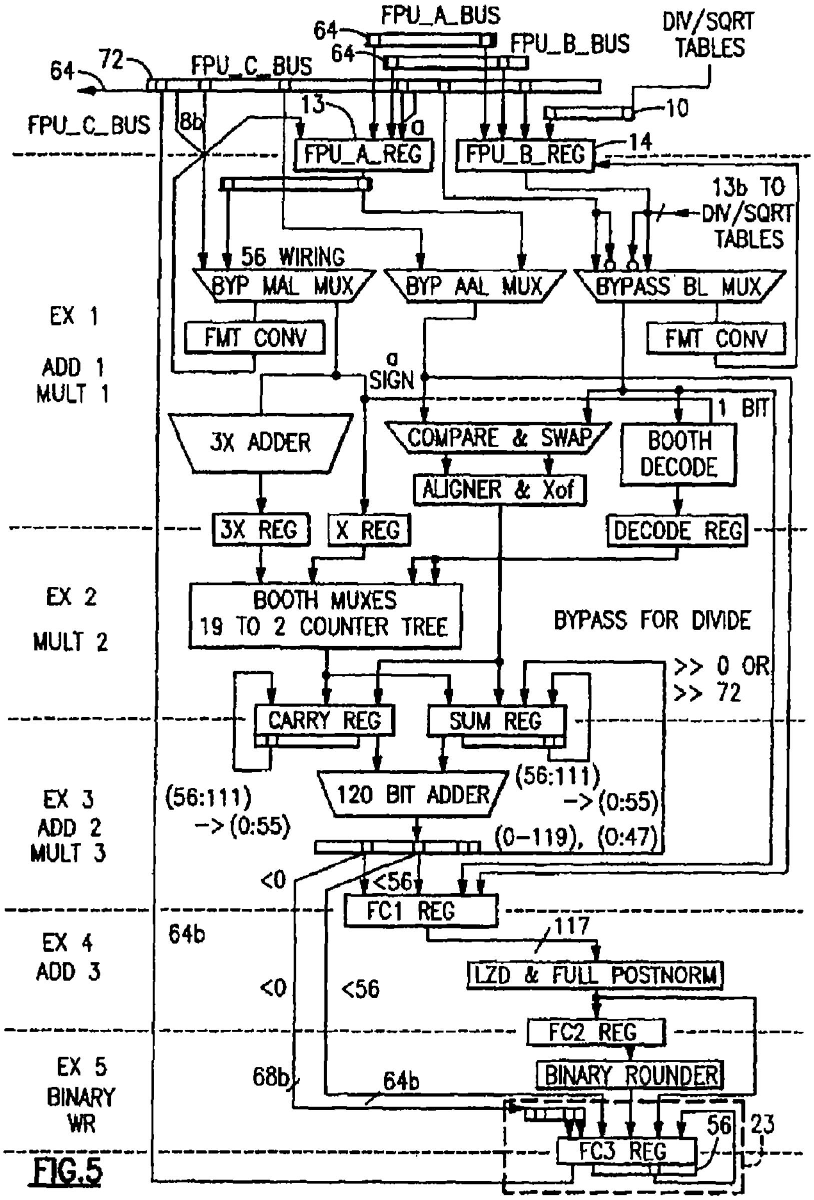

FROM A OR B REGISTERS : I_ = A_ OR B_
(I_EXP(0) || I_EXP(8:13) I_FRACT(0:7)
I_(1:15)
[1 1' || I_(2:15)]
[1 1' 1' 1' || I_(2:11)]
[1 1' 1' 1' 1' 1' 1' 1' || I_(2:8)]
LENGTH
FORM EXP MUX
50
FEXP(14:15)
2W MUX
51
BEXP(13:14)
BINARY_MODE
52
OLD_EXP
58
FEXP(0:15)
FEXP(0:13)
I_EXP(0:13)
ADD 1 OR 2
53
MUX
54
OUT_EXP(0:13)
[ANY BIT 7 TIMES || I_(1:8)]
[ANY BIT 4 TIMES || I_(1:11)]
I_(1:15)
LENGTH
FORM EXP MUX
55
15 WAY AND
57
15 WAY NOR
56
EXP_ONES
EXP_ZEROS

FIG.3



CONTROLS
FRACT REGISTER
80
LZD/LEN
82
ROUND TABLE
84
INCREMENT
81
DECREMENT
83
TRUNC/INCR/DECR
85
<3 <2 <1 <0 >1 >2 >3
7 TO 1 MUX
86
OP LENGTH
87
ROUND_OUT
C REGISTER
88
S/L/X
LATCH
78
NORMALIZATION
OVERFLOW
UNDERFLOW
BIN/HEX
EXP REGISTER
70
MULTIPLEXOR
REV SIGN EXT
71
LZD/LEN
72
+1 TO -6
15 BIT ADD
73
OP LENGTH
74
OVERWRITE FRACT
75
EXP_OUT
C EXP REGISTER
76

FIG.4



FPU_A_BUS
64
FPU_B_BUS
64
DIV/SQRT TABLES
64
72
FPU_C_BUS
FPU_C_BUS
8b
13
a
10
FPU_A_REG
FPU_B_REG
14
13b TO DIV/SQRT TABLES
56 WIRING
BYP MAL MUX
BYP AAL MUX
BYPASS BL MUX
EX 1
ADD 1
MULT 1
FMT CONV
FMT CONV
a
SIGN
1 BIT
3X ADDER
COMPARE & SWAP
BOOTH DECODE
ALIGNER & Xof
3X REG
X REG
DECODE REG
EX 2
MULT 2
BOOTH MUXES
19 TO 2 COUNTER TREE
BYPASS FOR DIVIDE
>> 0 OR
>> 72
CARRY REG
SUM REG
EX 3
ADD 2
MULT 3
(56:111)
-> (0:55)
120 BIT ADDER
(56:111)
-> (0:55)
(0-119), (0:47)
<0
<56
FC1 REG
117
EX 4
ADD 3
64b
<0
<56
LZD & FULL POSTNORM
FC2 REG
EX 5
BINARY
WR
68b
64b
BINARY ROUNDER
56
23
FC3 REG

FIG.5

IMPLEMENTATION OF BINARY FLOATING POINT USING HEXADECIMAL FLOATING POINT UNIT

FIELD OF THE INVENTION

The present invention relates to computer systems and architecture, and more particularly, to a computer system supporting two floating point architectures.

BACKGROUND OF THE INVENTION

In the ensuing description of the prior art and the present invention, the following are herein incorporated by reference:

"Enterprise Systems Architecture/390 Principles of Operation," Order No. SA22-7201-02, available through IBM branch offices, 1994; and

"IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754-1985," The Institute of Electrical and Electronic Engineers, Inc., New York, August 1985.

Generally, computer systems are designed in accordance with supporting a specific floating point architecture, such as either IEEE Binary Floating Point Standard 754-1985 or IBM S/390 hexadecimal floating point format. Generally, applications which are directed to one floating point standard are not compatible with another floating point standard. Thus, a user is limited in terms of "buying into" a certain architecture and to software applications developed therefor. Users, however, generally wish to run the same application on different platforms supporting different floating point architectures, or use data generated by applications on one platform with a different application on another platform that supports a different floating point architecture. Moreover, a user generally wishes to be able to run on a single machine applications that are based on different floating point architectures. These needs are particularly germane to multi-tasking environments provided by mainframe systems, where task switching occurs among many different users who generally wish to run applications based on different floating point architectures. Accordingly, the system must dynamically accommodate any of the different floating point architectures.

There is a need, therefore, to have a computer system that supports two floating point architectures. Moreover, such a computer system should provide the two floating point architectures with high performance and with limited resources. That is, multiple floating point architecture support should be provided without having to essentially duplicate floating point hardware and without sacrificing performance. In addition, such an architecture should generally be transparent to a user to the greatest extent possible in order to simplify interaction (e.g., programming, switching between different applications, etc.) between the user and the system.

SUMMARY OF THE INVENTION

The present invention provides a computer system supporting multiple floating point formats. In an embodiment of the present invention, an IBM S/390 hexadecimal floating point architecture and IEEE 754 binary floating point architecture are supported by having an internal dataflow which accommodates both formats, and in which all operations are executed. The system includes format conversion means for converting between the internal dataflow and the architected datatypes. In addition, in order to conform to the IEEE 754 standard, the system includes: means for determining a sticky bit; means for rounding a binary floating point number according to IEEE 754 rounding modes; and means for special number handling. In accordance with an embodiment of the present invention, the internal dataflow is a hexadecimal dataflow which optimizes the conversions from architected to internal format for the S/390 architecture.

BRIEF DESCRIPTION OF THE DRAWINGS

Additional aspects, features, and advantages of the invention will be understood and will become more readily apparent when the invention is considered in the light of the following description made in conjunction with the accompanying drawings, wherein:

FIG. 1 provides an overview of the dataflow of a Floating Point Unit (FPU), in accordance with the present invention;

FIG. 2 describes the fraction portion of binary architected to hex internal converters, in accordance with an embodiment of the present invention;

FIG. 3 shows the hardware employed for the exponent portion of the two binary architected to hex internal converters, in accordance with an embodiment of the present invention;

FIG. 4 shows the dataflow of a rounder and the format conversion to binary architected state, in accordance with and embodiment of the present invention; and

FIG. 5 further illustrates a dataflow in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

Supported Architectures

An implementation of the present invention described hereinbelow supports both S/390 hexadecimal floating point and IEEE 754 binary floating point numbers. The formats are shown below in Table 1:

Table 1

Supported Formats					
		Exponent (e)			Total
Format	Sign(s)	bits	bias	Fraction (f)	Width
Hex Short	1	7	64	24	32
Hex Long	1	7	64	56	64
Hex Extended	1	7	64	112	128
Binary Single	1	8	127	24	32
Binary Double	1	11	1023	53	64
Binary Double Extended	1	15	16383	113	128

Hexadecimal Floating point numbers have an exponent with a base of 16 as described by the following equation:

$$A=(-1)^{s}*f*16^{(e-bias)}$$

where f<1.0 and a normalized number is when f has the following characteristic: $0.1_{16}<=f<1.0_{16}$

Binary floating point numbers have an exponent with a base of 2 as described by the following equation:

$$A=(-1)^{s}*f*2^{(e-bias)}$$

where f<1.0 and a normalized number is when f has the following characteristic: $1.0_{2}<=1.f<10.0_{2}$

Also IEEE 754 standard defines several special numbers. These numbers are described in Table 2:

Table 2

IEEE 754 Special Numbers

Fraction	f = 0	f != 0
Exponent e = zeros	+0/−0	DeNorm
e = ones	+infinity/−infinity	NaN

If the fraction is zero and the exponent is all zeros, a plus or minus zero is represented. If the fraction is zero and the exponent is all ones, plus or minus infinity is represented. If the fraction is not equal to zero and the exponent is all zeros, a denormalized number is represented. If the exponent is all ones and the fraction is non-zero a Not-A-Number is represented.

HEX Internal Dataflow

In accordance with the present invention, providing support for two floating point architectures may be provided by having an internal dataflow which executes all floating point operations using an expanded format that accommodates both floating point formats. In an embodiment of the present invention, IEEE 754 and IBM S/390 hexadecimal formats are both supported by an expanded hexadecimal format to support IEEE 754 floating point numbers. This internal hexadecimal format supports the full exponent range of both IEEE 754 and S/390 with a 14 bit hexadecimal exponent biased by 8192. The dataflow supports a long floating point format which allows for 56 bits of fraction for each operand. The following is a description of this format:

$$A=(-1)^s \cdot f \cdot 16^{(e-8192)}$$

This format is very similar to the standard S/390 format but with a greater range of exponent. There is no leading one and the fraction is less than one. The bias is a hex type of bias (2^N) rather than a binary bias (2^N-1). This optimizes, in terms of latency, the conversions from architected to internal format for the S/390 architecture. It may be appreciated, however, that in certain situations it may instead be desirable to select an internal format that optimizes the conversions from architected to internal format for the IEEE 754 architecture.

Overview of Hardware Support for Multiple Architectures

The actual hardware needed to support both architectures consists of 4 types of format conversion hardware, a rounder, sticky bit determination, and controls for special number handling.

FIG. 1 provides an overview of the dataflow of the Floating Point Unit (FPU) which supports both S/390 and IEEE 754 architecture. The FPU has a localized high-speed memory referred to as the Floating Point Registers (FPRs) 10. The control unit which is not illustrated receives an instruction which it decodes and issues control signals to the dataflow. Instructions consist of an operation and a register specification. A typical instruction would indicate multiply FPR '0' by FPR '2' and place the result in FPR '0' ('0' and '2' refer to specific registers from among FPRs 10). Also, there is a mode bit which indicates whether to perform this operation in S/390 architecture or in IEEE 754. In addition, there is indication of the length of the operands in the instruction text. An instruction can also specify the operation to be performed on one register value and one value from memory with the result stored in a register. For simplicity this case is not shown but this case is supported by a memory bus being connected to B register 14.

Basically, two operands are transmitted from the FPRs 10 to the two hex architected to hex internal converters 11, 12. This conversion is rather straight forward given the representation of hex architected (56 fraction bits, 7 exponent bits, 1 sign bit, with hex exponent bias of 64) to hex internal notation (56 fraction bits, 14 exponent bits, 1 sign bit, with hex exponent bias of 8192), and is shown in Table 3 hereinbelow. The expansion is very similar to sign extension of a two's complement number. This format conversion is very fast and does not require much hardware. Thus, no extra stages are required for this conversion which is performed in parallel on both input operands.

There is a multiplexor 25 which drives the converter 11, its selection is controlled by the FPU control logic (not illustrated) which chooses between two input ports, the port 1 of the FPRs 10 and the result bus output from hex internal to hex architected block 24. The hex architected to hex internal converter 11 receives the output of multiplexor 25, converts the data to hex internal format, and transmits its output to the multiplexor 27 above A register 13. This multiplexor 27 chooses between the converter output 11 and the converter output 15. The choice is made by FPU control logic and is dependent on the binary mode and whether there is valid data in either converter. The output of multiplexor 27 is driven to A register 13.

There is a multiplexor 26 which drives the converter 12, its selection is controlled by the FPU control logic (not illustrated) which chooses between two input ports, the port 2 of the FPRs 10 and the result bus output from hex internal to hex architected block 24. The hex architected to hex internal converter 12 receives the output of multiplexor 26, converts the data to hex internal format, and transmits its output to the multiplexor above B register 28. This multiplexor 28 chooses between the converter output 12 and the converter output 16. The choice is made by FPU control logic and is dependent on the binary mode and whether there is valid data in either converter. The output of multiplexor 28 is driven to B register 14.

A register 13 receives the data and either transmits the result in the following cycle to the hex internal dataflow 18 if the operand was in hex format or transmits it to a binary architected to hex internal converter 15 which then transmits the result back to A register 13 and in the following cycle is transmitted on the hex internal dataflow 18.

B register 14 receives the data and either transmits the result in the following cycle to the hex internal dataflow 18 if the operand was in hex format or transmits it to a binary architected to hex internal converter 16 which then transmits the result back to B register 14 which and in the following cycle transmits the result on to the hex internal dataflow 18.

The binary architected to hex internal converters 15, 16 are described in more detail in FIG. 2 and FIG. 3.

Once the data has been converted to the hex internal representation which consists of a 56 bit fraction, 14 bit hexadecimal exponent biased by 8192, and a sign bit, the two operands are input to the hex internal dataflow 18 of the floating point unit. This hex internal dataflow 18 supports all floating point operations defined by S/390 and IEEE 754 such as add and multiply. The hex internal dataflow 18 is adapted to provide the function of computing a sticky bit 19 as defined by the IEEE 754 specification. This consists of ORing any bits after the guard bit that are truncated from the calculations which typically occurs on alignment of operands normalization of operands.

The output of the internal dataflow 18 is transmitted to a Rounder 22 and the C multiplexor 29. If the operation is on binary operands, the data is rounded 20 and transformed back into binary architected notation 21 by the rounder 22 and then transmitted to C multiplexor 29. The path skipping the rounder 22 and directly driving the data from the internal dataflow 18 of the C multiplexor 29 is used for hex operations. The C register 23 is driven by the C multiplexor 29 which is controlled by the FPU control logic which bases its selection on the mode of operations: hex or binary.

The output of C register 23 is connected to a hex internal to hex architected converter 24 simply involves transmitting the sign and fraction without any transformation and driving bits 0 and 8 through 13 of the 14 bit exponent. The output of this converter 24 is connected back to the FPRs to a write port and to wrap back to the dataflow to a multiplexor 25, 26 above the hex architected to hex internal converters 11, 12.

In addition, there is additional hardware included in the controls which are invoked in the case of a special number being detected in the binary architected to hex internal converters 15, 16. In this case, the pipeline of instructions is halted and the result is determined from a look-up table and transmitted to the FPRs. An example is multiplying positive infinity by a negative non-zero number which would result in the special number negative infinity being written into the FPRs.

Format Conversion: HEX Architected to HEX Internal

The transformation between these formats is shown by the following equations. Note that only the exponent needs to be changed.

$$(-1)^{sign}fa*16^{(Ea-64)}=(-1)^{s}fa*16^{(Ea^*-8192)}$$

$$16^{(Ea-64)}=64^{(Ea^*-8192)}$$

$$Ea-64=Ea^*-8192$$

$$Ea^*=Ea+8192-64$$

$$Ea^*=Ea+2^{13}-2^{6}$$

The following table shows that the transformation of exponent requires driving the most significant bit to bit position 0 and inverted to bit positions 1 through 7. Bits 1 through 6 of the input exponent are simply driven to bits 8

through 13 of the output exponent. Note that ' is used to indicated an inversion or complement operation.

TABLE 3

Format Conversion for Hex Architected to Hex Internal

Positional Weight	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
Bit Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Ea								x0	x1	x2	x3	x4	x5	x6
$2^{13}-2^{6}$	0	1	1	1	1	1	1	1	0	0	0	0	0	0
Ea* =	x0	x0'	x0'	x0'	x0'	x0'	x0'	x0'	x1	x2	x3	x4	x5	x6

Format Conversion: Binary Architected to HEX Internal

A binary number is transformed to have a hexadecimal exponent. This requires two less bits of exponent (base 2 versus base 16) and allows the hexadecimal datapath to be used for both formats. The number of exponent bits for double precision binary is 11 bits which requires only 9 bits for a hex exponent to represent. Note, that shifting and exponent update is needed to force the binary fraction into the hex fraction range. Also, the biases between the two architectures differ significantly. The following equations show how the final transformation is derived.

$$1.xxx*2^{(E-b)}=(\text{hex fraction})*16^{(E^*-b^*)}$$

Assume: E mod 4 = 0; b = 127 b* = 8192

$$1.xxx*2^{(E-127)}=(\text{hex-frac})*16^{(E^*-8192)}$$

$$=1.xxx*16^{((E-127)/4)}$$

$$=1.xxx*16^{((E-128)+1)/4+8192-8192)}$$

$$=1.xxx*2^{1}*16^{((E/4-32+8192)-8192)}$$

$$=1x.xxx*16^{((E/4+2^{(13)}-2^{(5)})-8192)}$$

$$=0.001xxx*16^{((E/4+1+2^{(13)}-2^{(5)})-8192)}$$

$$E^*=(E>>2)+1+2^{13}-2^{(5)}$$

$$E^*=SIGNEXT(E>>2)+1$$

Note that E>>2 is used to indicate a two bit shift right operation which is equivalent to an integer divided by 4. The result of E>>2 for the binary single notation has only 6 bits of significance. $2^{(13)}-2^{(5)}$ is equivalent to a series of ones from second most significant bit to the most significant bit of E>>2. Thus, the following describes this transformation:

Positional Weight	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
Bit Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$2^{13}-2^{5}$	0	1	1	1	1	1	1	1	1	0	0	0	0	0
E>>2									x8	x9	x10	x11	x12	x13
$(E>>2)+2^{13}-2^{5}$ =	x8	x8'	x8'	x8'	x8'	x8'	x8'	x8'	x8'	x9	x10	x11	x12	x13

This function of complementing the most significant bit and extending it to bit 0 of the 14 notation with the most significant bit no complemented (true) is defined to be the function SIGNEXT. It can be shown that this function occurs in the derivation for binary short, long, or extended format conversion. Below is a table describing the conversion for any of operand length:

$$1.xxx*2^{3}*2^{(E-bias)}=(0.0001_2\|52\text{ bits})*16^{(SIGNEXT(E>>2)+2)-8192)}$$

$1.xxx*2^0*2^{(E-bias)}=(0.001_2||52\ bits||0)*16^{(SIGNEXT(E>>2)+1)-8192)}$

$1.xxx*2^1*2^{(E-bias)}=(0.01_2||52\ bits||00)*16^{(SIGNEXT(E>>2)+1)-8192)}$

$1.xxx*2^2*2^{(E-bias)}=(0.1_2||52\ bits||000)*16^{(SIGNEXT(E>>2)+1)-8192)}$

The implementation of conversion and detection of special numbers is shown and described in additional detail in FIG. 2 and FIG. 3.

FIG. 2 describes the fraction portion of binary architected to hex internal converters, 15, 16 from FIG. 1. These converters 15, 16 are the same and rather than using A or B register for describing the input signals, instead the signal "I_" representing the sign, exponent, and fraction, and "I_SIGN" representing only the sign bit, "I_EXP" represents the 14 exponent bits, and "I_FRACT" represents the 64 fraction bits. Note that there are 64 fraction bits maintained internally to provide higher precision on intermediate calculations such an example is the division operation where a higher precision must be maintained for intermediate calculations than for the input or final result.

I_(0:71)=I_SIGN||I_EXP(0)||I_EXP(8:13)

|| I_FRACT(0:63) where || represents the concatenation symbol and (M:N) represents range of bits of the input from bit M to and including bit N.

Note that this fraction converter can be separated into two functions: the fraction conversion and the detection of special numbers. First the fraction conversion is described.

I_(0:71) is input to the 5 to 1 multiplexor 30 where any of 5 different formats are chosen based on the input operand length. For a short operand the format FMT_S is selected, for a long operand the format FMT_L is chosen, for extended operands with 113 bits of fraction, the data is partitioned into 3 groups. The high order bits have the format FMT_XH, the middle or low bits have the format FMT_XL, and the very least significant or sometimes called guard bits have the format FMT_XG. These formats are described by the following equations.

FMT_S(0:59)={"1"||I_(9:31)||0(36)}

FMT_L(0:59)={"1"||I_(23:63)||0(7)}

FMT_XH(0:59)={"1"||I_(16:63)||I_(0:6)||0(4)}

FMT_XL(0:59)={I_(4:63)}

FMT_XG(0:59)={0(48)||I_(60:63)||0(8)}

where "1" represents a logical value of one or high, and 0(M) represents the concatenation of M number of logical zeros or low values.

For the extended operands, the operations are performed by a software routine which operates on the 3 portions of data.

The most significant bit, bit 0, of the multiplexor 30 is transmitted to a 2 way AND gate 31. The other input to the multiplexor is a signal which indicates that the number is not denormalized which is formed by a 2 input NAND gate 32. This NAND gate 32 has two inputs exp_zeros which comes from the paired exponent portion of the hex architected to hex internal converter and a control signal called binary_mode which indicates the operation is in binary mode if it is a "1" (high) and indicates hex mode if it is a "0" (low). These gates effectively cancel the implied "1" in binary notation if the exponents indicate that the number is denormalized number. The output bit 0 of the AND gate 31 and the lesser significant bits of the output bit 0 of the AND gate 31 and the lesser significant bits of the output of the multiplexor 30 form the signal BFRAC (0:59). Bits 1 to 59 are transmitted to special number detector in addition to all the bits being used to form 7 formats of 59 bits. The first format is shift 0, the second is a shift 1 bit right with a fill of the most significant bit with a "0" (low) signal. The fourth format is a shift three bits right and a three bit "0" fill. The fifth format is to shift 1 bit left and fill with "0". The sixth format is to shift 2 bit left and fill with "0". The seventh format is to shift 3 bit left and fill with "0". These seven formats are input to the data ports of the 7 to 1 multiplexor 33.

The exponent least significant bits, bits 13 and 14, indicated the binary alignment within the hexadecimal notation that fraction should use. These two bits are called BEXP (13:14) and are input from the exponent portion of the binary architected to hex internal format convert and are received by the selection shift logic 34. Also the operand format is input to this block from controls. The following is a table of the selection depending on BEXP and the operand format:

BEXP			
13	14	FORMAT	SELECT
0	0	S, L, XH	shift right by 2
0	1	S, L, XH	shift right by 1
1	0	S, L, XH	shift zero
1	1	S, L, XH	shift right by 3
0	0	XL, XG	shift left by 1
0	1	XL, XG	shift left by 2
1	0	XL, XG	shift left by 3
1	1	XL, XG	shift zero

Thus, the seven to 1 multiplexor 33 produces the output fraction from having the shift selection outputs 34 and the seven possible shifts.

The special number determination logic receives the inputs BFRAC(1:59) from the fraction portion of this figure and exp_ones and exp_zeros from the exponent portion of this converter, and a binary_mode signal. BFRAC(1:59) is input to a 59 way NOR gate 35 which determines if the fraction is zero, if so then the output is a "1" This output is driven to 2 way AND gates 36 and 37 and 2 way AND gates with this input inverted 37 and 39. Also, input to gates 36 and 37 is exp_zeros, and also input to gates 38 and 39 is exp_ones. The output of gate 36 indicates that the special number zero has been received and the signal out_zero is sent to the special number handling controls 17 in FIG. 1. The output of gate 37 indicates that the special number denormalized has been received and the signal out_denorm is sent to the special number handling controls 17 in FIG. 1. The output of gate 38 indicates that the special number infinity has been received and the signal out_inf is sent to the special number handling controls 17 in FIG. 1. The output of gate 39 indicates that the special number Not-A-Number (NAN) has been detected and a further elaboration of whether it is a quite or signaling NaN is needed. Thus, 3 input AND gate 40 with the input BFRAC(1) inverted, as the

true inputs binary_mode and the output of gate 39 is used to determine a signaling NaN which is called out_snan and output to the special number handling controls 17 in FIG. 1. The 3 input AND gate 41 with the inputs BFRAC(1), binary_mode, and the output of gate 39 is used to determine a quiet NaN which is called out_qnan and output to the special number handling controls 17 in FIG. 1.

FIG. 3 shows the hardware employed for the exponent portion of the two binary architected to hex internal converters 15, 16. In the exponent portion, only "I_" bits 1 to 15 are needed. Three formats are created for the three different binary exponent formats: EXP_S, EXP_L, EXP_X; for short, long, and extended operands respectively.

EXP_S={I_(1)||I_(1)'||I_(1)'||I_(1)'||I_(1)'||I_(1)'||I_(1)'||I_(1)'||I_(1)'||I_(2:8)}

EXP_L={I_(1)||I_(1)'||I_(1)'||I_(1)'||I_(1)'||I_(1)'||I_(2:11)}

EXP_X={I_(1)||I_(1)'||I_(2:15)}

where I_(1)' indicate the complementation of I_(1).

The 3 input Format Exponent Multiplexer (form Exp Mux) 50 receives these 3 formats as input. The multiplexer 50 also receives the length signal from FPU controls. If short is indicated by the length signal EXP_S is selected, if long then EXP_L is selected. If extended then EXP_X is selected. The output of this multiplexer 50 is represented as FEXP (0:15). The most significant 14 bits, FEXP (0:13) are driven to the adder 53 and the multiplexer 54. The least significant 2 bits of FEXP, bits 14 and 15, are driven to the multiplexer 51 and also a register 58. The output of register 58 is called OLD_EXP(14:15) and is delayed one cycle from the signal FEXP(14:15). The multiplexer 51 is driven by OLD_EXP(14:15) and FEXP(14:15). FPU Controls determines the selection signals for this 2 to 1 multiplexer 51 which is dependent on the format desired. If the format is S, L, or XH, then FEXP(14:15) is selected, and if the format is XL or XG then OLD_EXP(14:15) is selected. The output of this multiplexer 51 is called BEXP(13:14) and it is driven to the control block 52 which determines the selection of variable adder 53 and selection of the 3 input multiplexer 54. This control block also receives an input the signal binary_mode from the FPU controls. If the binary_mode signal is low then the operation is in hex mode and multiplexer 54 chooses the original I_EXP(0:13) as output and the variable adder can add 1 or 2 to the FEXP(0:13) which is a don't care situation. If the binary mode signal is high then the variable adder 53 and multiplexor 54 are set up to produce the following results depending upon the BEXP(13:14):

BEXP(13:14)	OUT_EXP(0:13)	Variable Add 53	Mux Select 54
00	FEXP(0:13) + 1	+1	Variable Add Out
01	FEXP(0:13) + 1	+1	Variable Add Out
10	FEXP(0:13) + 1	+1	Variable Add Out
11	FEXP(0:13) + 2	+2	Variable Add Out

The control block 52 select outputs for the variable adder 53 and the multiplexor 54 can be determined from this table for binary mode.

The Variable Adder 53 receives as input FEXP(0:13) from the multiplexor 50. It also receives a select signal from control block 52 which determines whether it should add 1 or 2 to its other input. The output of the adder 53 is driven to multiplexor 54. Multiplexor 54 receives as input the output of the variable adder 53, the input to this overall exponent format converted, I_EXP(0:13) and it receives the output of the multiplexor 50, FEXP(0:13). Other modes of operation, such as handling special numbers can select some of the other combinations not given in the table above which overrides the selection from control block 52. The output of multiplexor 54 is the output exponent, OUT_EXP(0:13).

Also part of the exponent format converter is the detection of special numbers. Special numbers in IEEE 754 standard have all the exponent bits zeros or all ones. To detect this for all three formats which can have 8 exponent bits for short, 11 exponent bits for long, and 15 exponent bits for extended; all three formats are expanded to a 15 bit format. To do this three input formats are created: I_(1:15), any bit of I_(1:11) replicated 4 times and concatenated to I_(1:11), and the third format is any bit of I_(1:8) replicated 7 times and concatenated to I_(1:8). These three formats are input to format multiplexor 55, which chooses the first format if the length signal indicates extended format, the second format for long length, and the third format for short length. The output of the 3 to 1 multiplexor 55 is output to a 15 way NOR gate 56 and a 15 way AND gate 57. The NOR gate 56 determines if all the exponent bits are zero and transmits an active high or 1 signal called exp_zeros if this is the case, else a 0 or low signal is transmitted. The receiver of this signal is the fraction format convert described in FIG. 2. The AND gate 57 determines if all the exponent bits are all ones and transmits the signal exp_ones to the fraction format convert described in FIG. 2.

Rounding and Format Conversion of Hex Internal to Binary Architected

The conversion back to binary from hex is done in the rounder and involves the following formulation:

$$
\begin{aligned}
&0.hfrac * 16^{(B-8192)} = 1.xxx * 2^{(B'' - b'')}\\
&\text{Assume: } hfrac = .1xxx;\ b'' = 1023\\
&0.1xx * 16^{(B-8192)} = 1.xxx * 2^{(B'' - 1023)}\\
&= 1.xxx * 2^{(-1)} * 2^{(4*(B-8192))}\\
&= 1.xxx * 2^{(4B - 32768 - 1)}\\
&= 1.xxx * 2^{((4B - 32768 + 1024 - 2) - 1023)}\\
&B'' = 4 * B - 2^{(15)} + 2^{(10)} - 2\\
&B'' = REVSIGNEXT(4*B) - 2
\end{aligned}
$$

The function REVSIGNEXT can be illustrated by the following table:

$$1.xxx*2^{0}*2^{(E-bias)}=(0.001_2\|52\ \text{bits}\|0)*16^{(SIGNEXT(E>>2)+1)-8192)}$$

$$1.xxx*2^{1}*2^{(E-bias)}=(0.01_2\|52\ \text{bits}\|00)*16^{(SIGNEXT(E>>2)+1)-8192)}$$

$$1.xxx*2^{2}*2^{(E-bias)}=(0.1_2\|52\ \text{bits}\|000)*16^{(SIGNEXT(E>>2)+1)-8192)}$$

The implementation of conversion and detection of special numbers is shown and described in additional detail in FIG. 2 and FIG. 3.

FIG. 2 describes the fraction portion of binary architected to hex internal converters, 15, 16 from FIG. 1. These converters 15, 16 are the same and rather than using A or B register for describing the input signals, instead the signal "I_" representing the sign, exponent, and fraction, and "I_SIGN" representing only the sign bit, "I_EXP" represents the 14 exponent bits, and "I_FRACT" represents the 64 fraction bits. Note that there are 64 fraction bits maintained internally to provide higher precision on intermediate calculations such an example is the division operation where a higher precision must be maintained for intermediate calculations than for the input or final result.

I_(0:71)=I_SIGN||I_EXP(0)||I_EXP(8:13)

|| I_FRACT(0:63) where || represents the concatenation symbol and (M:N) represents range of bits of the input from bit M to and including bit N.

Note that this fraction converter can be separated into two functions: the fraction conversion and the detection of special numbers. First the fraction conversion is described.

I_(0:71) is input to the 5 to 1 multiplexor 30 where any of 5 different formats are chosen based on the input operand length. For a short operand the format FMT_S is selected, for a long operand the format FMT_L is chosen, for extended operands with 113 bits of fraction, the data is partitioned into 3 groups. The high order bits have the format FMT_XH, the middle or low bits have the format FMT_XL, and the very least significant or sometimes called guard bits have the format FMT_XG. These formats are described by the following equations.

FMT_S(0:59)=["1"||I_(9:31)||0(36)]

FMT_L(0:59)=["1"||I_(23:63)||0(7)]

FMT_XH(0:59)=["1"||I_(16:63)||I_(0:6)||0(4)]

FMT_XL(0:59)=[I_(4:63)]

FMT_XG(0:59)=[0(48)||I_(60:63)||0(8)]

where "1" represents a logical value of one or high, and 0(M) represents the concatenation of M number of logical zeros or low values.

For the extended operands, the operations are performed by a software routine which operates on the 3 portions of data.

The most significant bit, bit 0, of the multiplexor 30 is transmitted to a 2 way AND gate 31. The other input to the multiplexor is a signal which indicates that the number is not denormalized which is formed by a 2 input NAND gate 32. This NAND gate 32 has two inputs exp_zeros which comes from the paired exponent portion of the hex architected to hex *internal* converter and a control signal called binary_mode which indicates the operation is in binary mode if it is a "1" (high) and indicates hex mode if it is a "0" (low). These gates effectively cancel the implied "1" in binary notation if the exponents indicate that the number is denormalized number. The output bit 0 of the AND gate 31 and the lesser significant bits of the output bit 0 of the AND gate 31 and the lesser significant bits of the output of the multiplexor 30 form the signal BFRAC (0:59). Bits 1 to 59 are transmitted to special number detector in addition to all the bits being used to form 7 formats of 59 bits. The first format is shift 0, the second is a shift 1 bit right with a fill of the most significant bit with a "0" (low) signal. The fourth format is a shift three bits right and a three bit "0" fill. The fifth format is to shift 1 bit left and fill with "0". The sixth format is to shift 2 bit left and fill with "0". The seventh format is to shift 3 bit left and fill with "0". These seven formats are input to the data ports of the 7 to 1 multiplexor 33.

The exponent least significant bits, bits 13 and 14, indicated the binary alignment within the hexadecimal notation that fraction should use. These two bits are called BEXP (13:14) and are input from the exponent portion of the binary architected to hex *internal* format convert and are received by the selection shift logic 34. Also the operand format is input to this block from controls. The following is a table of the selection depending on BEXP and the operand format:

BEXP			
13	14	FORMAT	SELECT
0	0	S, L, XH	shift right by 2
0	1	S, L, XH	shift right by 1
1	0	S, L, XH	shift zero
1	1	S, L, XH	shift right by 3
0	0	XL, XG	shift left by 1
0	1	XL, XG	shift left by 2
1	0	XL, XO	shift left by 3
1	1	XL, XG	shift zero

Thus, the seven to 1 multiplexor 33 produces the output fraction from having the shift selection outputs 34 and the seven possible shifts.

The special number determination logic receives the inputs BFRAC(1:59) from the fraction portion of this figure and exp_ones and exp_zeros from the exponent portion of this converter, and a binary_mode signal. BFRAC(1:59) is input to a 59 way NOR gate 35 which determines if the fraction is zero, if so then the output is a "1" This output is driven to 2 way AND gates 36 and 37 and 2 way AND gates with this input inverted 37 and 39. Also, input to gates 36 and 37 is exp_zeros, and also input to gates 38 and 39 is exp_ones. The output of gate 36 indicates that the special number zero has been received and the signal out_zero is sent to the special number handling controls 17 in FIG. 1. The output of gate 37 indicates that the special number denormalized has been received and the signal out_denorm is sent to the special number handling controls 17 in FIG. 1. The output of gate 38 indicates that the special number infinity has been received and the signal out_inf is sent to the special number handling controls 17 in FIG. 1. The output of gate 39 indicates that the special number Not-A-Number (NAN) has been detected and a further elaboration of whether it is a quite or signaling NaN is needed. Thus, 3 *input* AND gate 40 with the input BFRAC(1) inverted, as the

true inputs binary_mode and the output of gate 39 is used to determine a signaling NaN which is called out_snan and output to the special number handling controls 17 in FIG. 1. The 3 input AND gate 41 with the inputs BFRAC(1), binary_mode, and the output of gate 39 is used to determine a quiet NaN which is called out_qnan and output to the special number handling controls 17 in FIG. 1.

FIG. 3 shows the hardware employed for the exponent portion of the two binary architected to hex internal converters 15, 16. In the exponent portion, only "L_" bits 1 to 15 are needed. Three formats are created for the three different binary exponent formats: EXP_S, EXP_L, EXP_X; for short, long, and extended operands respectively.

EXP_S={L_(1)||L_(1)'||L_(1)'||L_(1)'||L_(1)'||L_(1)'||L_(1)'||L_(1)'||L_(1)'||L_(1)'||L_{0}(2:8)}

EXP_L={L_(1)||L_(1)'||L_(1)'||L_(1)'||L_(1)'||L_(1)'L_(2:11)}

EXP_X={L_(1)||L_(1)'||L_(2:15)}

where L_(1)' indicate the complementation of L_(1).

The 3 input Format Exponent Multiplexer (form Exp Mux) 50 receives these 3 formats as input. The multiplexer 50 also receives the length signal from FPU controls. If short is indicated by the length signal EXP_S is selected, if long then EXP_L is selected, if extended then EXP_X is selected. The output of this multiplexer 50 is represented as FEXP (0:15). The most significant 14 bits, FEXP (0: 13) are driven to the adder 53 and the multiplexer 54. The least significant 2 bits of FEXP, bits 14 and 15, are driven to the multiplexer 51 and also a register 58. The output of register 58 is called OLD_EXP(14:15) and is delayed one cycle from the signal FEXP(14:15). The multiplexer 51 is driven by OLD_EXP(14:15) and FEXP(14:15). FPU Controls determines the selection signals for this 2 to 1 multiplexer 51 which is dependent on the format desired. If the format is S, L, or XH, then FEXP(14:15) is selected, and if the format is XL or XG then OLD_EXP(14:15) is selected. The output of this multiplexer 51 is called BEXP(13:14) and it is driven to the control block 52 which determines the selection of variable adder 53 and selection of the 3 input multiplexer 54. This control block also receives an input the signal binary_mode from the FPU controls. If the binary_mode signal is low then the operation is in hex mode and multiplexer 54 chooses the original I_EXP(0:13) as output and the variable adder can add 1 or 2 to the FEXP(0:13) which is a don't care situation. If the binary mode signal is high then the variable adder 53 and multiplexor 54 are set up to produce the following results depending upon the BEXP(13:14):

BEXP(13:14)	OUT_EXP(0:13)	Variable Add 53	Mux Select 54
00	FEXP(0:13) + 1	+1	Variable Add Out
01	FEXP(0:13) + 1	+1	Variable Add Out
10	FEXP(0:13) + 1	+1	Variable Add Out
11	FEXP(0:13) + 2	+2	Variable Add Out

The control block 52 select outputs for the variable adder 53 and the multiplexor 54 can be determined from this table for binary mode.

The Variable Adder 53 receives as input FEXP(0:13) from the multiplexor 50. It also receives a select signal from control block 52 which determines whether it should add 1 or 2 to its other input. The output of the adder 53 is driven to multiplexor 54. Multiplexor 54 receives as input the output of the variable adder 53, the input to this overall exponent format converted, I_EXP(0:13) and it receives the output of the multiplexor 50, FEXP(0:13). Other modes of operation, such as handling special numbers can select some of the other combinations not given in the table above which overrides the selection from control block 52. The output of multiplexor 54 is the output exponent, OUT_EXP(0:13).

Also part of the exponent format converter is the detection of special numbers. Special numbers in IEEE 754 standard have all the exponent bits zeros or all ones. To detect this for all three formats which can have 8 exponent bits for short, 11 exponent bits for long, and 15 exponent bits for extended; all three formats are expanded to a 15 bit format. To do this three input formats are created: L_(1:15), any bit of L_(1:11) replicated 4 times and concatenated to L_(1:11), and the third format is any bit of L_(1:8) replicated 7 times and concatenated to L_(1:8). These three formats are input to format multiplexor 55, which chooses the first format if the length signal indicates extended format, the second format for long length, and the third format for short length. The output of the 3 to 1 multiplexor 55 is output to a 15 way NOR gate 56 and a 15 way AND gate 57. The NOR gate 56 determines if all the exponent bits are zero and transmits an active high or 1 signal called exp_zeros if this is the case, else a 0 or low signal is transmitted. The receiver of this signal is the fraction format convert described in FIG. 2. The AND gate 57 determines if all the exponent bits are all ones and transmits the signal exp_ones to the fraction format convert described in FIG. 2.

Rounding and Format Conversion of Hex Internal to Binary Architected

The conversion back to binary from hex is done in the rounder and involves the following formulation:

$$0.hfrac * 16^{(B-8192)} = 1.xxx * 2^{(B'' - b'')}$$
Assume: hfrac = .1xxx; b'' = 1023
$$0.1xx * 16^{(B-8192)} = 1.xxx * 2^{(B'' - 1023)}$$
$$= 1.xxx * 2^{(-1)} * 2^{(4*(B-8192))}$$
$$= 1.xxx * 2^{(4B - 32768 - 1)}$$
$$= 1.xxx * 2^{((4B - 32768 + 1024 - 2) - 1023)}$$
$$B'' = 4 * B - 2^{(15)} + 2^{(10)} - 2$$
$$B'' = REVSIGNEXT(4*B) - 2$$

The function REVSIGNEXT can be illustrated by the following table:

Positional Weight	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
Bit Position	−2	−1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
E =			x5	x5'	x5'	x5'	x5'	x5'	x6	x7	x8	x9	x10	x11	x12	x13
4E =	x5	x5'	x5'	x5'	x5'	x5'	x6	x7	x8	x9	x10	x11	x12	x13	0	0
$2^{10}-2^{13}$	0	−1	−1	−1	−1	−1	0	0	0	0	0	0	0	0	0	0
$4E-2^{13}+2^{10}$	0	0	0	0	0	x5	x6	x7	x8	x9	x10	x11	x12	x13	0	0

Thus, the following table can be derived:

(0.0001|xxx)*16^(E−8192)=1.xxx*2^([illegible])

(0.001|xxx)*16^(E−8192)=1.xxx*2^([illegible])

(0.01|xxx)*16^(E−8192)=1.xxx*2^([illegible])

(0.1|xxx)*16^(E−8192)=1.xxx*2^([illegible])

The dataflow of the rounder 22 of FIG. 1 and the format conversion to binary architected state is shown and described in FIG. 4. The output sign, exponent, faction, guard bit, and sticky bit of the hex internal dataflow 18 are transmitted to the Rounder exponent register 70, fraction register 80 and to controls where they are latched up and in the following cycle drive to other blocks to be described. The exponent is driven to a multiplexor 71 which performs reversing of the sign extension. The output of this multiplexor is called REXP(0:14) and the input can be represented by a generic signal called IN_EXP(0:13). The following table indicates the selection made by the multiplexor block 71:

binary_mode	length	REXP(0:14)
0	—	[0 || IN_EXP(0:13)]
1	short	[0(7) || IN_EXP(0) || IN_EXP(9:13) || "00"]
1	long	[0(4) || IN_EXP(0) || IN_EXP(6:13) || "00"]

-continued

binary_mode	length	REXP(0:14)
1	extended	[IN_EXP(0) || IN_EXP(2:13) || "00"]

The output of the multiplexor 71, REXP(0:14), is driven to the variable adder 73 which also receives a selection of how much to add from control block 72. Control block 72 receives input from a latch indicating to decrement or increment from latch 78 and its outputs are called RDECR_EXP_Q which indicates to decrement, and RINCR_EXP_Q which indicates to increment when active high. Also binary mode and the most significant four bits of the fraction are received by control block 72. This control block determines which bit in the fraction has the most significant one which is guaranteed by the dataflow to be in bit 0, 1, 2, or 3 if the result is non-zero. The following is decisions made by this control block 72 based on these inputs where dash indicates a don't care state:

RDECR_EXP_Q	RINCR_EXP_Q	BINARY_MODE	LZD	ADD_OUT(0:14)
0	0	0	—	REXP + 0
1	0	0	—	REXP − 1
0	1	0	—	REXP + 1
0	0	1	0	REXP − 2
0	0	1	1	REXP − 3
0	0	1	2	REXP − 4
0	0	1	3	REXP − 5
1	0	1	0	REXP − 3
1	0	1	1	REXP − 4
1	0	1	2	REXP − 5
1	0	1	3	REXP − 6
0	1	1	0	REXP − 1
0	1	1	1	REXP − 2
0	1	1	2	REXP − 3
0	1	1	3	REXP − 4
1	1	x	x	REXP + 0

Thus, the selection signals for the 15 bit variable adder 73 are determined by control block 72. The variable adder is capable of adding 1 or 0, or subtracting 1, 2, 3, 4, 5, or 6. The output of the adder is called ADD_OUT(0:14) and it is driven to operand length multiplexer 74. The operand length multiplexer receives control signals to select between several possible formats of ADD_OUT(0:14) or is able to force a constant. The possible constants are dependent on the special number controls and the signal rnd_zero and rnd_inf is used to force a special number in the rounder. The following is a table of the controls:

BINARY_MODE	RND_ZERO	RND_INF	Length	GEXP(0:14)
0	0	0	—	[ADD_OUT(8)' || ADD_OUT(9:14) || 0(8)]
1	0	0	short	[ADD_OUT(7:14) || 0(7)]
1	0	0	long	[ADD_OUT(4:14) || 0(4)]
1	0	0	extended	[ADD_OUT(0:14)]
—	1	0	—	"000000000000000"
—	0	1	—	"111111111111111"
—	1	1	—	"000000000000000"

The most significant 7 bits, GEXP(0:6), is driven to the overwrite fraction multiplexor where a choice is made between GEXP(0:6) and RNDF_OVERWR(1:7) which is signal from the fraction rounder multiplexor 86 which contains fraction bits which may overwrite exponent and sign bit. If the format is extended and it is the second write cycle, the RNDF_OVERWR(1:7) is chosen, else GEXP (0:6) is chosen to drive to C exponent register 76.

The fraction dataflow involves incrementing and decrementing the fraction in parallel with determining the rounding. The fraction is driven from register 80 to an incrementer 81 and a decrementor 83. These are not normal incrementers and decrementers since the dataflow has fight justified the data into the least significant bits of the fraction register but the justification can be off by 0, 1, 2, or 3 depending on the implied binary exponent which is implied within the hex normalization of the number. Thus, a leading zero detect of the most significant bits is used to determine the location of the least significant bit. If the first four bits are "0001" for an extended format the bit to increment and decrement is the least significant on register 80. If the first three bits are "0001" for an extended format the next more significant bit has the weight of the appropriate increment or decrement position. The increment 81 and decrementer 83 are 116 bits and receive the location of the bit to increment from the control block 82. The control block 82 determines the location as follows:

BINARY_MODE	Length	LZD	Location	LSB	Guard Bit	Sticky Bits
Hex	Short	x	112	112	113	x
Binary	Short	3	115	115	116	sticky
Binary	Short	2	114	114	115	sticky or 116
Binary	Short	1	113	113	114	sticky or (116 to 115)
Binary	Short	0	112	112	113	sticky or (116 to 114)
Hex	Long	x	112	112	113	x
Binary	Long	3	115	115	116	sticky or (117 to 119)
Binary	Long	2	114	114	115	sticky or (116 to 119)
Binary	Long	1	113	113	114	sticky or (115 to 119)
Binary	Long	0	112	112	113	sticky or (114 to 119)
Hex	Extended	x	112	112	113	x
Binary	Extended	3	115	115	116	sticky
Binary	Extended	2	114	114	115	sticky or (116)
Binary	Extended	1	113	113	114	sticky or (115 to 116)
Binary	Extended	0	112	112	113	sticky or (114 to 116)

Also, shown in this table is the determination of which bit is the least significant, which bit is the guard bit, and which bits to or together in addition to the bits which were ORed together in the hex dataflow which is represented by the signal, sticky.

These signals are transmitted on to the rounding Table 84 and are not necessary to determine in control block 82 if they were determined completely in the hex internal dataflow. In the example implementation, only partial sticky information is provided by the hex internal dataflow which includes sticky bit calculation to the hex digit boundary but not within the digit which depends on the leading zero detect.

The rounding table 84 is used to determine whether the fraction should have been rounded up, down or truncated. This determines the select signals for the 3 input multiplexor 85. The rounding table follows the conventions dictated by the IEEE 754 standard which determine whether round up, down, or truncate based on the rounding mode and the least significant bit (LSB), the guard bit (G), and the sticky bits (S). For instance if the rounding mode is round to nearest and the following is the rounding dictated:

LSB	G	S	Choose:
x	0	x	truncate
x	1	1	increment
0	1	0	truncate
1	1	0	increment

This is a standard implementation of an IEEE 754 rounding table. The output signals of the rounding table 84 drive the 3 to 1 multiplexor 85 which receives outputs from the fraction register 80 if truncate is selected, the incrementer 81 if increment is selected, and the decrementer 83 if decrement is selected. Additionally, the carry out and selection are used to determine if the exponent needs to be incremented or decremented in addition to any other type of correction. If there is a carry out of the incrementer 81 and increment is selected then a signal is driven to latch 78 indicating that an additional increment of the exponent is needed. If there is no carry out of decrementer and decrement is selected then an additional decrement of the exponent is needed and this indication is driven to latch 78. Control logic will take over in these rare cases to force the proper result using this dataflow. The normally chosen path is that neither of these cases are indicated to latch 78 and the data output of multiplexor 85 is driven to the 7 to 1 multiplexor 86 which can shift up to 3 bits to the right or 3 bits to the left. The

selection of how many bits to shift is determined by control block 82. There are multiple modes that control block 82 can base its selection on such as binary align the data in the case of a divide operation or the case that concerns this invention the transforming of hex internal data to binary architected. In this case the data is shifted based on the examination of the most significant digit of the fraction. If the first bit of this most significant digit is a one, a shift of zero is indicated; if the most significant two bits of this digit are "01" then a shift of 1 bit to the left is indicated. If the most significant three bits of this digit are "001" then a shift of 2 bits to the left is indicated, and neither of these cases are indicated then a shift of 3 bits to the left is indicated.

The output of this multiplexor 86 is driven to the fraction operand length multiplexor 87 along with the output of the exponent rounder GEXP(7:14). The following are selections of this multiplexor 87 which are determined by the operand length, and mode from controls:

Binary Mode	Length	ROUND_OUT(0:55)
Hex	Short	FRACT(89:112) || 0(32)
Hex	Long	FRACT(57:112)
Hex	Extended cycle 1	FRACT(0:55)
Hex	Extended cycle 2	FRACT(56:111)
Binary	Short	GEXP(7) || FRACT(90:112) || 0(32)
Binary	Long	GEXP(7:10) || FRACT(61:112)
Binary	Extended cycle 1	GEXP(7:14) || FRACT(1:48)
Binary	Extended cycle 2	FRACT(57:112)

Thus, ROUND_OUT is determined which drives the fraction C register 88. Also, output from multiplexer 86 is the most significant 8 bits to potentially overwrite the sign and exponent for the first cycle of an extended binary operand, this signal is referred to as RNDF_OVERWR (0:7).

Miscellaneous Hardware

As described above, sticky bit calculation is also a part of the overall scheme for executing binary floating point format on a hexadecimal dataflow. Commonly assigned U.S. patent application Ser. No. 08/414,072, filed Mar. 31, 1995, and entitled "Parallel Calculation of Exponent and Sticky Bit During Normalization" describes an embodiment of sticky bit hardware that may be employed in the present invention.

Special Number Handling controls, as described hereinabove, are also included as part of a preferred embodiment of the present invention. They are tables that determine either the result given that one or both of the input operands is a special number, or they determine an execution routine. The first mechanism is simply a look-up table which contains the result based on the type of number for each input operand and the type of instruction being executed. An example of this mechanism is that given an input operand is a NaN (Not-A-Number) the result is equal to a NaN. The second mechanism is similar to a micro routine. It determines the control signals for the multiple cycles of execution of the operation. an example of this mechanism is a routine to calculate an binary add of denormalized numbers. The dataflow is put in a non-pipelined mode state and then the operands are prenormalized prior to adding them. An ordinary hexadecimal dataflow would calculate a wrong result if prenormalization was not performed. This second mechanism determines the control signals for each cycle of execution until the result is written to the FPRs.

EXAMPLES

The following instructions, including Add Long and Multiply Long, are provided as examples to illustrate operation of the dataflow according to the present invention which is not limited thereto. In connection with the description of these instructions, FIG. 5 provides further illustration of functional elements included in the dataflow according to an embodiment of the present invention.

Add Long (not special)

Each cycle of the add long is described and an example is used for clarity of exposition. For an example assume:

$$
\begin{aligned}
A &= +1.0 \\
&= (-1)^0*(1).0000*2^{1023-1023} \\
A\_SIGN &= 0 \\
A\_EXP &= 011,1111,1111_2 = 3FF_{16} \\
A\_FRACT &= 0 \\
A\_INPUT &= 3FF0000000000000_{16} \\
B &= -0.FFFFFFFFFFFFF8_{16} \\
&= (-1)^1*(1).FFFFFFFFFFFFF_{16}*2^{1022-1023} \\
B\_SIGN &= 1 \\
B\_EXP &= 011,1111,1110_2 = 3FE_{16} \\
B\_FRACT &= FFFFFFFFFFFFF_{16} \\
B\_INPUT &= BFEFFFFFFFFFFFFF_{16}
\end{aligned}
$$

1. The dataflow for a binary add long involves format conversion, using the normal add dataflow, and then rounding. First the operands are latched into A and B registers 13 and 14, respectively, assuming that they are hex operands. The only problem with this is that there are some exponent bits in the most significant bits of fraction. Thus, the original 64 bit input is recreated from examining the sign, exponent, and fraction registers.

A(0:63)=A_SIGN_REG_Q‖A_EXPREG_Q(0,3:13)‖A_REG_Q(0:55)

The binary fraction is formed by taking ("1"‖A(12:63)‖0(7)) where "1" denotes a binary one and 0(n) denotes n concatenated binary zeros. This is transformed into a hex fraction by binary aligning it using bits A(10:11) to determine the alignment. The resulting hex fraction is 56 bits since at most 3 zeros need to appended to the most significant bits. The hex exponent is also calculated by the format converter 11. A(1:11) is SIGNEXT to 14 bits after the least significant two bits are truncated. Then one is added to resulting exponent and additional one is added using the carry in to the adder 53 if A(10:11)==3(BEXP 13:14; which corresponds to adding 2). Thus, the new exponent is easy to form. The format converters also check the input operands to see if they are special numbers. If not, the following is the add procedure.

Using the same example data, format conversion is performed:

$$
\begin{aligned}
A\_SIGN\_REG\_Q &= 0 \\
EXP \bmod 4 &= 3 \\
A\_EXPREG\_Q(0{:}13) &= SIGNEXT(3FF_{16}>>2)+2 \\
&= 01,1111,1111,1111_2+2 \\
&= 10,0000,0000,0001_2 = 2001_{16} \\
A\_REG\_Q(0{:}55) &= (0.0001 \,\|\, f) \\
&= 10000000000000_{16} \\
B\_SIGN\_REG\_Q &= 1 \\
EXP \bmod 4 &= 2 \\
B\_EXPREG\_Q(0{:}13) &= SIGNEXT(3FE_{16}>>2)+1 \\
&= 01,1111,1111,1111_2+1 \\
&= 10,0000,0000,0000_2 = 2000_{16} \\
B\_REG\_Q(0{:}55) &= (0.1 \,\|\, f \,\|\, 000) \\
&= FFFFFFFFFFFFF8_{16}
\end{aligned}
$$

2. The 56 bit fractions and 14 bit exponents are sent to the first cycle of a three cycle pipeline for add which involves

aligning the fractions. The exponents are compared and the operand with the smaller exponent is shifted to the alignment of the larger exponent. The smaller operand is also complemented if the operation is an effective subtraction. The alignment results in a 60 bit fraction for the small operand which maintains an addition hex digit of guard bits. The aligner is paired with a hardware unit located in sticky bit block 19 which determines if the alignment caused any shifting out of the 60 bits which are equal to one. This is needed later for the rounding circuit. Note that shifting out ones for an add will result in the calculated value being less than the actual, but for an effective subtract the calculated value will be more than the actual. At the end of the A1 cycle the 60 bit small operand and 56 bit big operand are latched in low order bits of a SUM and a CARRY register starting at bit 56. Note that if a zero operand is detected, the alignment can be overridden. The exponent of the big operand is latched into the SC_EXPREG.

For the example:

SC_EXPREG_Q(0:13)=10,0000,0000,0001$_2$=2001$_{16}$

SUM_REG_Q(0:119)=(00000000000000‖10000000000000‖00)$_{16}$

SUM_REG_Q(0:119)=(0000000000000010000000000000000)$_{16}$

CARRY_REG_Q(0:119)=(FFFFFFFFFFFFFF‖0FFFFFFFFFFFFF8‖F)$_{16}$

CARRY_REG_Q(0:119)=(FFFFFFFFFFFFFF‖F00000000000007‖F)$_{16}$

CARRY_REG_Q(0:119)=(FFFFFFFFFFFFFFFF00000000000007F)$_{16}$

CARRY_IN_REG=1

STICKY_BIT=0

3. The next cycle involves performing the carry propagate add (2 to 1 add). The result of the add is 60 bits with potentially a carry. The result is shifted 56 bits to line up with bit 0 of FC1 register and the carry (add out bit 55) is latched into FC1 Carry register. The exponent is moved from the SC_EXPREG to the FC1_EXPREG in this cycle. No bits of significance are lost during this add so no further sticky bit calculation is needed in this cycle.

For the example:

SUM_REG_Q(0:119)=(0000000000000010000000000000000)$_{16}$

CARRY_REG_Q(0:119)=(FFFFFFFFFFFFFFFF00000000000007F)$_{16}$

ADD_OUT(0:119)=(000000000000000000000000000080)$_{16}$

output is shifted 56b

FC1_REG_Q(0:119)=0000000000000008000000000000000)$_{16}$

FC1_EXPREG_Q(0:13)=10,0000,0000,0001$_2$=2001$_{16}$

STICKY BIT=0

4. The fourth cycle of execution involves normalizing the resulting data. A Leading Zero Detect (LZD) of the fraction is performed to determine the number of hex digits to shift left. The fraction that is shifted consists of a carry digit (3 zeros and the carry register) and the fraction digits. The exponent in FC1 corresponds to a radix point between the carry and fraction digits. The exponent during normalization is first incremented by 1 to correspond to a radix point to the left of the carry digit and then the shift amount is subtracted from the exponent. The resulting fraction is latched into FC2 register and exponent in FC2_EXPREG. A sticky bit calculation needs to be performed in this cycle to determine if there is no shift and then see if the last hex digit of significance contains any ones.

For example:

FC1_REG_Q(0:119)=(0000000000000008000000000000000)$_{16}$

FC_1EXPREG_Q(0:13)=10,0000,0000.0001$_2$=2001$_{16}$

FC2_REG_Q(0:115)=(80000000000000000000000000000)$_{16}$

FC2_EXPREG_Q(0:13)=01,1111.1111,0011$_2$=1FF3$_{16}$

STICKY_BIT=0

5. The fifth cycle of execution performs rounding and format conversion. The last bits of the least significant hex digit are examined for ones depending on in which location in the most significant hex digit there is a one. The fraction minus these last bits is aligned and then sent to a decrementer 83 and rounder 20. The sticky information and rounding mode is used to access a look-up table which determines whether the result should be truncated, incremented, or decremented. And a 3 to 1 multiplexor is used to choose the appropriate result. Additionally, a format conversion 21 is performed which deletes the most significant one and updates the exponent with the binary normalization information. The format conversion merges the sign, exponent, and fraction and stores them into FC3 sign, exponent, and fraction (i.e., C register 23) registers as though the data were hex. This implies that only 7 of the exponent bits are stored in the exponent register and the rest are merged with the faction and stored in FC3 register. Thus, rounding and format conversion are performed in this fifth cycle of execution.

For the example:

FC2_REG_Q(0:115)=(80000000000000000000000000000)$_{16}$

FC2_EXPREG_Q(0:13)=01,1111,1111,0011$_2$=1FF3$_{16}$

STICKY BIT=0

BFRAC=$(-1)^0 * 0.10000_2 * 16^{1FF3_{16}-2000_{16}}$

SIGN=0

FRAC=(1).000000$_2$ one is implied

EXP(0:10)=REVSIGNEXT(4*01,1111,1111,0011$_2$)−2

EXP(0:10)=REVSIGNEXT(0111,1111,1100,1100$_2$)−2

EXP(0:10)=011,1100,1100$_2$)−2

EXP(0:10)=011,1100,1010$_2$)=970$_{10}$

result=$+1.0*2^{970-1023}=2^{-53}$

Multiply Long (not special)

To illustrate the steps of the multiply long the same data from add long is used.

A =	+1.0
	= $(-1)^0*(1).0000*2^{1023-1023}$
A_SIGN =	0
A-EXP =	011,1111,1111$_2$ = 3FF$_{16}$
A_FRACT =	0
A_INPUT =	3FF0000000000000$_{16}$
B =	−3.1 0.FFFFFFFFFFFFF8$_{16}$
	= $(-1)^1*(1)$FFFFFFFFFFFFF$_{16}*2^{1022-1023}$
B_SIGN =	1
B_EXP =	011,11111,1110$_2$ = 3FE$_{16}$
B_FRACT =	FFFFFFFFFFFFF$_{16}$
B_INPUT =	BFEFFFFFFFFFFFFF$_{16}$

1. Whether the operation be multiply or add, the binary data must be converted into hex data. Thus, the first cycle of

execution involves format conversion. This is the same as for add and is not pipelinable, but both operands are converted in parallel and stored back into A and B registers.

Using the same example data, format conversion is performed:

A_SIGN_REG_Q = 0
EXP mod 4 = 3
A_EXPREG_Q(0:13) = SIGNEXT(3FF$_{16}$ >>2) + 2
= 01,1111,1111,1111$_2$ +2
= 10,0000,0000,0001$_2$ =2001$_{16}$
A_REG_1(0:55) = (0.0001 ||f)
= 10000000000000$_{16}$
B_SIGN_REG_Q = 1
EXP mod 4 = 2
B_EXPREG_Q(0:13) = SIGNEXT(3fe$_{16}$ >>2) + 1
= 01,1111,1111,1111$_2$ +1
= 10,0000,0000,0001$_2$ =2000$_{16}$
B_REG_Q(0:55) = (0.1 ||f||000)
= FFFFFFFFFFFFF8$_{16}$

2. The hex internal dataflow has a three cycle pipeline multiplication. The second cycle of executing the multiplication involves executing the first cycle of the hex dataflow pipeline which involves a 3× calculation of operand A and 4 bit overlap Booth decoding operand B. All fraction bits participate and none are shifted out. The A operand 3× calculation and staging is latched in 3× and X registers which have two copies each called A copy and B copy. The Booth decode logic is latched in the Booth Decode registers. The exponent calculation consists of adding the two exponents and subtracting the bias, 8192, and the result is latched in the M1 EXPREG.

For the example:

M1_EXPREG = 2001$_{16}$ + 2000$_{16}$ − 2000$_{16}$(bias)
= 2001$_{16}$
X_REG_1(0:63) = 1000000000000000$_{16}$
3X_REG_Q(0:65) = 0300000000000000$_{16}$

3. The second cycle of execution in the hex internal pipeline involves completing the summing of partial products using a counter tree. The result is two terms called the carry and sum which are latched in the CARRY and SUM register. The exponent is M1 EXPREG is moved to SC EXPREG in this cycle.

For the example:

SC_EXPREG=2001$_{16}$

The result is two terms called carry and sum.

4. The fourth cycle involves a carry propagate add of the CARRY and SUM register and the resulting fraction is latched in FC1 register. The fraction can be hex unnormalized by at most one hex digit if the input data was not zero or a denormalized number. The exponent is driven from SC EXPREG to FC1 EXPREG in this cycle.

For the example:

FC1_EXPREG=2001$_{16}$

ADD_OUT(0:119) = 0FFFFFFFFFFFFF8000000000000000$_{16}$

FC1_REG$_{1,1}$(0:119) = 0FFFFFFFFFFFFF8000000000000000$_{16}$

5. The fifth cycle is not really necessary but it eases the control design of the dataflow. This cycle normalizes the data using a full post normalizer. The resulting fraction is latched into FC2 register (i.e., registers to 180). In this cycle, the sticky bit is calculated based on any hex digits which extend beyond the length of the format (short=7

hex digits, long=14 hex digits). The exponent is updated with the shift amount which could at most be one hex digit shift left. The resulting exponent is stored into FC2 EXPREG.

For the example:

FC2_EXPREG=2000$_{16}$

FC2_REG_Q(0:115) = FFFFFFFFFFFFF8000000000000000$_{16}$

6. The final cycle involves rounding 20 and format conversion 21. The sticky bit and rounding mode are used to determine whether to increment or truncate the 53 bit product after binary alignment. The corresponding exponent is also updated. The final 53 bits of fraction and 8 bits of exponent along with the sign bit are converted into a hex format which is ready to be written in architectural format.

For the example:

FC2_EXPREG=2000$_{16}$

FC2_REG_Q(0:115) = FFFFFFFFFFFFF8000000000000000$_{16}$

BFRAC=$(-1)^{1}$*0.FFFFFFFFFFFFF8$_{16}$*$16^{2000_{16}-2000_{16}}$

SIGN=1

(0.1||xxx)*16^{E-8192}=1.xxx*$2^{REVSIGNEXT(4*E)-2-bias}$

EXP=REVSIGNEXT(4*10.0000.0000.0000$_2$)−2

EXP=REVSIGNEXT(1000,0000,0000,0000$_2$)−2

EXP=100,0000,0000$_2$−2

EXP=011,1111,1110$_2$=3FE$_{16}$=1022$_{10}$

result=$(-1)^{1}$*(1).FFFFFFFFFFFFF$_{16}$*$2^{1022-1023}$

result=$(-1)^{1}$*(1).FFFFFFFFFFFFF$_{16}$*2^{-1}

result=−0.FFFFFFFFFFFFF8$_{16}$

FC3 register is assumed to be in hex format and it is converted into architectural format (conversion only involves wiring concatenation) and is driven onto the FPU C bus (i.e., result bus). This bus drives the register files FPRs 10.

It may be appreciated, therefore, that the proposed invention provides a means for executing floating point operations in either of two architected floating point data types, and particularly, either S/390 hexadecimal format or IEEE 754 binary format. The present invention provides a floating point unit having very high performance on hex operations and provides compatibility of binary operations, without significant loss of performance for these binary operations.

Although the above description provides many specificities, these enabling details should not be construed as limiting the scope of the invention, and it will be readily understood by those persons skilled in the art that the present invention is susceptible to many modifications, adaptations, and equivalent implementations without departing from this scope and without diminishing its attendant advantages. It is therefore intended that the present invention is not limited to the disclosed embodiments but should be defined in accordance with the claims which follow.

What is claimed is:

1. A computer system supporting a plurality of floating point architectures, each flowing point architecture having at least one format, said system comprising:

a floating point unit having an internal dataflow according to an internal floating point format having a number of exponent bits which is the minimum number required to support each of said plurality of floating point architectures, said floating point unit performing floating point operations in said internal floating point format; and

conversion means for converting between each one of said plurality of floating point architectures and said internal floating point format such that an operand of any one of said plurality of floating point architectures input to said floating point unit is converted into said internal floating point format for operation by said floating point unit, and the result of the operation is converted into any one of said plurality of floating point architectures.

2. The computer system according to claim 1, wherein said plurality of floating point architectures includes a hexadecimal floating point architecture and IEEE 754 binary floating point architecture.

3. The computer system according to claim 2, wherein said floating point unit includes:

means for determining a sticky bit;

means for rounding a binary floating point number according to IEEE 754 rounding modes; and

means for special number handling.

4. The computer system according to claim 2, wherein said hexadecimal floating point architecture is IBM S/390 hexadecimal architecture.

5. The computer system according to claim 4, wherein said internal floating point format is a hexadecimal format that has a fourteen bit exponent biased by 8192, a sign bit, and a 56 bit fraction.

6. The computer system according to claim 1, wherein said plurality of floating point architectures includes a binary architected format and an hexadecimal architected format, and said internal format is a hexadecimal internal format.

7. The computer system according to claim 6, wherein said hexadecimal architected format and said hexadecimal internal format each have a common predetermined bias type, said predetermined bias type being either an even bias or an odd bias.

8. The computer system according to claim 7, wherein said predetermined bias type is even, and said hexadecimal internal format has a nonzero positive integer number (N+1) of exponent bits and a bias equal to 2^N.

9. The computer system according to claim 1, wherein said plurality of floating point architectures includes a binary architected format and an hexadecimal architected format, and said internal format is a hexadecimal internal format, and wherein said conversion means includes:

a first converter that converts said hexadecimal architected format to said hexadecimal internal format;

a second converter that converts said binary architected format to said hexadecimal internal format;

a third converter that converts said hexadecimal internal format into said binary architected format; and

a fourth converter that converts said hexadecimal internal format into said hexadecimal architected format.

10. The computer system according to claim 1, wherein said plurality of floating point architectures includes a binary architected format and an hexadecimal architected format, and said internal format is a hexadecimal internal format, and wherein said conversion means includes:

a first converter that applies a transformation to convert said hexadecimal architected format to said hexadecimal internal format, and applies said transformation to binary architected format data to provide transformed binary architected format data;

a second converter that converts said transformed binary architected format data to said hexadecimal internal format;

a third converter that converts said hexadecimal internal format into said transformed binary architected format data; and

a fourth converter that applies a second tranformation that converts said hexadecimal internal format into said hexadecimal architected format, and applies said second transformation to said transformed binary architected format data to provide said binary architected format data.

11. In a computer system, a floating point unit that supports a first floating point architecture and a second floating point architecture, said first and second floating point architectures each having at least one format, said system comprising:

a floating point unit having an internal dataflow with an internal floating point format which supports both said first floating point architecture and said second floating point architecture, and that performs floating point operations on data formatted in said internal floating point format;

a first converter that applies a transformation to convert an operand of said first floating point architecture type to said internal floating point format, and applies said transformation to an operand of said second floating point architecture type to provide transformed second floating point architecture type data;

a second converter that converts said transformed second floating point architecture type data to said internal floating point format;

a third converter that converts data of said internal floating point format into data of said transformed second floating point architecture type; and

a fourth converter that applies a second tranformation that converts data of said internal floating point format into data of said first floating point architecture, and applies said second transformation to data of said transformed second floating point architecture type to provide data of said second floating point architecture type.

12. The computer system according to claim 11, wherein an operand of said first floating point architecture type is directly converted into said internal floating point format by said first converter for operation by said floating point unit, and wherein an operand of said second floating point architecture type is converted into said internal floating point format by said first converter and said second converter operating in series.

13. The computer system according to claim 11, wherein data in said internal floating point format is directly converted into data of said first floating point architecture by said fourth converter, and wherein data in said internal floating point format is converted into data of said second floating point architecture type by said third converter and said fourth converter operating in series.

14. The computer system according to claim 11, wherein said internal floating point format has a minimum nonzero positive number of exponent bits, N, to support both first and second floating point architectures.

15. The computer system according to claim 11, wherein said first and second floating point architectures have different bias types, the bias types being either of even type or odd type, said internal format having a bias type that matches said first floating point architecture.

16. The computer system according to claim 15, wherein said internal floating point format has a minimum nonzero positive number of exponent bits, N, to support both first and second floating point architectures, and wherein said internal floating point has an even bias of $2^{(N-1)}$.

17. The computer system according to claim 11, wherein said internal format has the same base and bias type as one of said first and second floating point architectures, the same bias type being either even or odd.

* * * * *

EXHIBIT 18

US005696709A

United States Patent [19]
Smith, Sr.

[11] Patent Number: 5,696,709
[45] Date of Patent: Dec. 9, 1997

[54] PROGRAM CONTROLLED ROUNDING MODES

[75] Inventor: Ronald Morton Smith, Sr., Wappingers Falls, N.Y.

[73] Assignee: International Business Machines Corporation, Armonk, N.Y.

[21] Appl. No.: 414,866

[22] Filed: Mar. 31, 1995

[51] Int. Cl.[6] G06F 7/38
[52] U.S. Cl. 364/745; 364/748
[58] Field of Search 364/745, 748, 364/715.04

[56] References Cited

U.S. PATENT DOCUMENTS

4,823,260 4/1989 Imel et al. 364/745 X

Primary Examiner—Tan V. Mai
Attorney, Agent, or Firm—Lynn L. Augspurger; David V. Rossi

[57] ABSTRACT

A computer system having a default floating point rounding mode that may be overridden by a rounding mode designated by an instruction. The current machine rounding mode is stored in a register, and an instruction includes a field for specifying whether rounding should be performed according to the current rounding mode or according to another rounding mode during execution thereof.

12 Claims, 3 Drawing Sheets

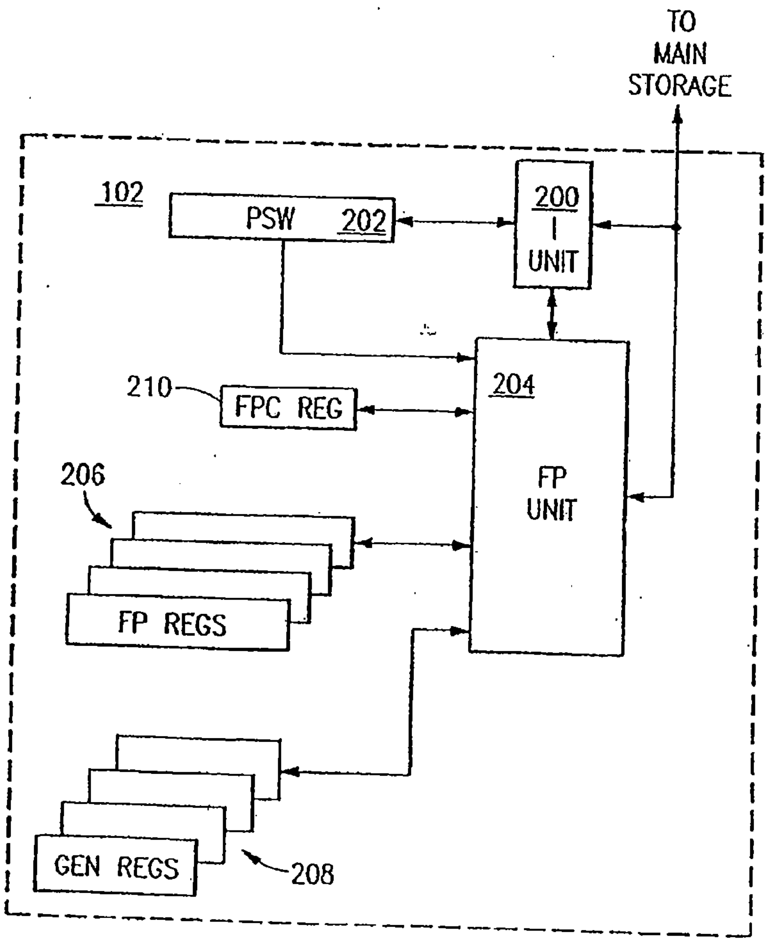


FIG.1

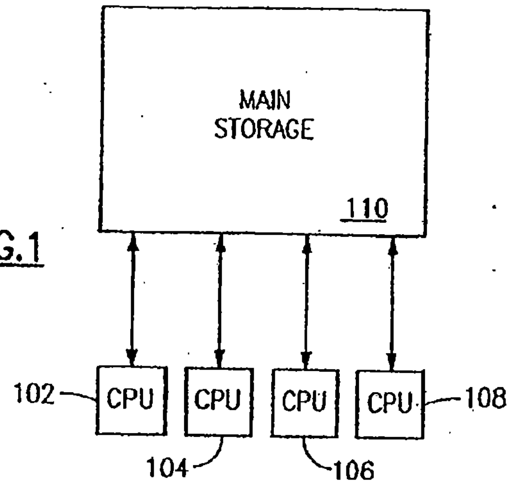
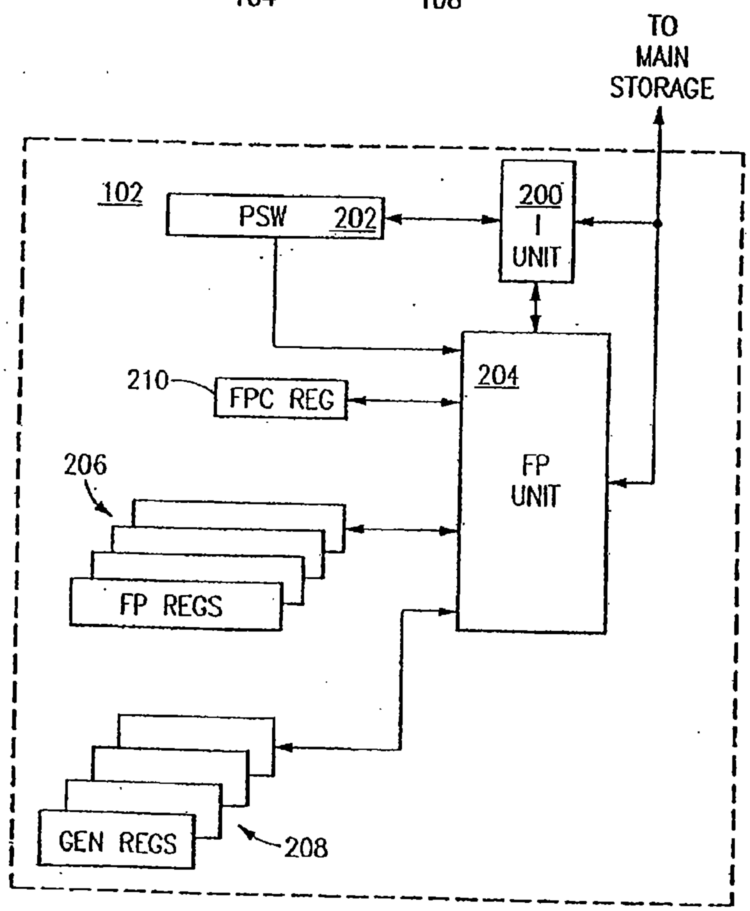

MAIN
STORAGE
110
102
CPU
CPU
CPU
CPU
108
104
106

FIG.2


TO
MAIN
STORAGE
102
PSW 202
200
I
UNIT
210
FPC REG
204
FP
UNIT
206
FP REGS
GEN REGS
208

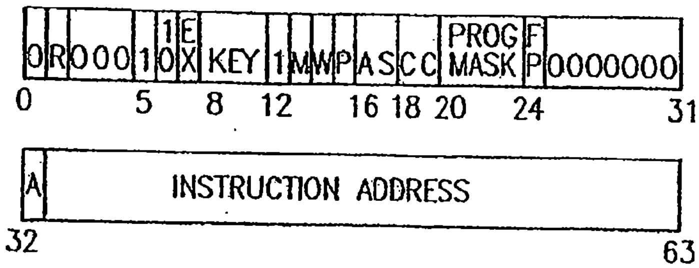


FIG.3

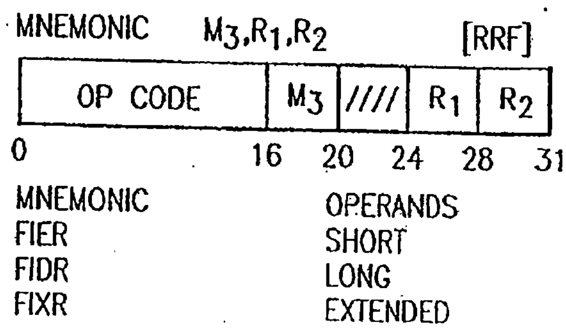


FIG.5

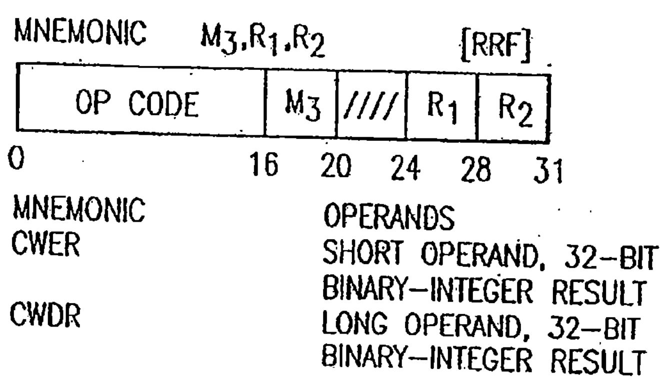


FIG.6

1

BYTE	BITS	VALUE	FUNCTION
0	0		BFP-INVALID-DATA MASK
	1		BFP-DIVISION-BY-ZERO MASK
	2		BFP-OVERFLOW MASK
	3		BFP-UNDERFLOW MASK
	4		BFP-INEXACT MASK
	5-7		(UNASSIGNED)
1	0		BFP-INVALID-DATA FLAG
	1		BFP-DIVISION-BY-ZERO FLAG
	2		BFP-OVERFLOW FLAG
	3		BFP-UNDERFLOW FLAG
	4		BFP-INEXACT FLAG
	5-7		(UNASSIGNED)
2	0-7		DATA-EXCEPTION CODE (DXC)
3	0		QNaN MODE
	1-5		(UNASSIGNED)
	6-7		ROUNDING MODE
		00	ROUND TO NEAREST
		01	ROUND TO ZERO
		10	ROUND UP
		11	ROUND DOWN

FIG.4

PROGRAM CONTROLLED ROUNDING MODES

FIELD OF THE INVENTION

The present invention relates to computer systems and, more particularly, to a computer architecture which includes instructions providing for programmable control of a rounding mode.

BACKGROUND OF THE INVENTION

In the ensuing description of the prior art and the present invention, the following are herein incorporated by reference:

"Enterprise Systems Architecture/390 Principles of Operation," Order No. SA22-7201-02, available through IBM branch offices, 1994;

"IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754-1985," The Institute of Electrical and Electronic Engineers, Inc., New York, August 1985; and

Commonly assigned U.S. patent application Ser. No. 08/414,250 to Eric Mark Schwarz, et al., filed Mar. 31, 1995, and entitled "Implementation of Binary Floating Point Using Hexadecimal Floating Point Unit".

In past architectures, rounding was provided either by means of a mode which controlled the rounding on all instructions, or by means of special rounding instructions. Each of these schemes has advantages and disadvantages. The mode has an advantage when a particular rounding mode is desired for an extended period of time. The special instructions have an advantage when a specific rounding is required for a single operation.

It would be advantageous, however, to have a machine which incorporates both a rounding mode and a rounding instruction.

SUMMARY OF THE INVENTION

The present invention overcomes the above, and other, prior art limitations by providing a machine having a default rounding mode that may be overridden by a rounding mode designated by an instruction. The current machine rounding mode is stored in a register, and an instruction includes a field for specifying whether rounding should be performed according to the current rounding mode or according to another rounding mode during execution thereof.

BRIEF DESCRIPTION OF THE DRAWINGS

Additional aspects, features, and advantages of the invention will be understood and will become more readily apparent when the invention is considered in the light of the following description made in conjunction with the accompanying drawings, wherein:

FIG. 1 illustrates a conventional shared memory computer system which may be employed to implement the present invention;

FIG. 2 schematically depicts functional components included in a CPU which may be employed in accordance with the present invention;

FIG. 3 illustrates the format of a 64 bit program status word (PSW), including a bit for indicating a binary or hexadecimal floating point mode, in accordance with an embodiment of the present invention;

FIG. 4 illustrates the format of a floating-point-control (FPC) register, including bits for indicating a rounding mode, in accordance with the present invention;

FIG. 5 illustrates the format of a LOAD FP Integer instruction, including a rounding mode field, in accordance with the present invention; and

FIG. 6 illustrates the format of a Convert to Fixed instruction, including a rounding mode field, in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

FIG. 1 illustrates a conventional shared memory computer system including a plurality of central processing units (CPUs) 102–108 all having access to a common main storage 110. FIG. 2 schematically depicts functional components included in a CPU from FIG. 1. Instruction unit 200 fetches instructions from common main storage 110 according to an instruction address located in the program status word (PSW) register 202, and appropriately effects execution of these instructions. Instruction unit 200 appropriately hands off retrieved floating point instructions to floating point unit 204, along with some of the operands that may be required by the floating point unit to execute the instruction. Floating point (FP) unit 204 includes all necessary hardware to execute the floating point instruction set, and preferably, in accordance with an embodiment of the present invention, supports both Binary and Hexadecimal floating point formats. FP unit 204 is coupled to floating point (FP) registers 206, which contain floating point operands and results associated with FP unit 204 processing, and is also coupled to general registers 208. FP unit 204 is also coupled to floating point control (FPC) register 210, which preferably includes mask bits in addition to those provided in the PSW, as well as bits indicating the floating point mode. In a multi-user application, FPC register 210 is under control of the problem state.

FIG. 3 illustrates the format of a 64 bit PSW as stored in PSW register 202. In a multi-user application, the supervisor state saves the PSW for a given problem state when taking interruption to dispatch another problem state. It can be seen that PSW includes program mask bits 20–23.

FP-Mode Bit in PSW

Bit 24 of the PSW is the FP-mode bit. In accordance with an embodiment of the present invention whereby both binary and hexadecimal floating point modes are supported, when the bit is zero, the CPU is in the hexadecimal-floating-point (HFP) mode, and floating-point operands are interpreted according to the HFP format. When the bit is one, the CPU is the binary-floating-point (BFP) mode, and floating-point operands are assumed to be in the BFP format. Some floating-point instructions operate the same in either mode.

When an instruction is executed which is not available in the current FP mode, a special-operation exception is recognized.

FPC Register

As illustrated in detail by FIG. 4, the floating-point-control (FPC) register 210 is a 32-bit register, which contains the mode (i.e., rounding mode), mask, flag, and code bits. For this implementation, by way of example, the rounding mode is represented by the last two bits of the last byte. Round to nearest, round to zero, round up, and round down modes are supported.

Program Controlled Rounding Modes

In accordance with the present invention, the rounding mode indicated by the FPC register 210 may be superceded by certain instructions that are executed. Two instructions, LOAD FP INTEGER and CONVERT TO FIXED, are provided as examples of an embodiment of implementing program controlled rounding modes according to the present

invention, which is not limited thereto. FIG. 5 illustrates the format of a LOAD FP Integer instruction which may be executed by FP Unit 204. Execution of this instruction results in a floating point number located in a FP register 206 identified by the second operand R_2 being rounded to an integer value in the same floating-point format, with the result placed in the first-operand location R_1 which identifies a floating point register 206. The resulting integer, which remains in floating-point format, either hexadecimal or binary, should not be confused with binary integers, which use a fixed-point format. If the floating-point operand is numeric with a large enough exponent so that it is already an integer, the

FPC Register

As illustrated in detail by FIG. 4, the floating-point-control (FPC) register 210 is a 32-bit register, which contains the mode (i.e., rounding mode), mask, flag, and code bits. For this implementation, by way of example, the rounding mode is represented by the last to bits of the last byte. Round to nearest, round to zero, round up, and round down modes are supported.

Program Controlled Rounding Modes

In accordance with the present invention, the rounding mode indicated by the FPC register 210 may be superceded by certain instructions that are executed. Two instructions, LOAD FP INTEGER and CONVERT TO FIXED, are provided as examples of an embodiment of implementing program controlled rounding modes according to the present invention, which is not limited thereto. FIG. 5 illustrates the format of a LOAD FP Integer instruction which may be executed by FP Unit 204. Execution of this instruction results in a floating point number located in a FP register 206 identified by the second operand R_2 being rounded to an integer value in the same floating-point format, with the result placed in the first-operand location R_1 which identifies a floating point register 206. The resulting integer, which remains in floating-point format, either hexadecimal or binary, should not be confused with binary integers, which use a fixed-point format. If the floating-point operand is numeric with a large enough exponent so that it is already an integer, the result value remains the same, except that, in the HFP mode, an unnormalized operand is normalized, and an operand with a zero fraction is changed to a true zero.

In accordance with an embodiment of the present invention, a modifier in the M_3 field controls the method of rounding in the BFP mode. The second operand, if numeric, is rounded to an integer value as specified by the modifier in the M_3 field:

M_3 Rounding Method

0 According to current rounding mode

1 Biased round to nearest

4 Round to nearest

5 Round to zero

6 Round up

7 Round down

When the modifier field is zero, rounding is controlled by the current rounding mode in the FPC register. When the field is not zero, rounding is performed as specified by the modifier, regardless of the current rounding mode. Rounding for modifiers 4–7 is the same as for rounding modes 0–3 (binary 00–11), respectively. Biased round to nearest (modifier 1) is the same as round to nearest (modifier 4), except when the second operand is exactly halfway between two integers, in which case the result for biased rounding is the next integer that is greater in magnitude. It may be understood that, in accordance with an embodiment of the present invention where both hexadecimal and binary floating point are supported, if the modifier is 5, the mehtod of rounding is the same in the HFP and BFP modes.

FIG. 6 illustrates the format of a Convert to Fixed instruction which may be executed by FP Unit 204. Execution of this instruction results in a floating point number located in a FP register 206 identified by the second operand R_2 being converted to a binary-integer, fixed-point format, with the result placed in the first-operand location R_1 which identifies a general register 208.

The result of CWDR and CWER is a 32-bit signed binary integer that is placed in the general register designated by R_1. A modifier in the M_3 field controls the method of rounding.

If the second operand is numeric, finite, and not already an integer, it is converted to an integer value in the fixed-point format by rounding as specified by the modifier in the M_3 field:

M_3 Rounding Method

0 According to current rounding mode

1 Biased round to nearest

4 Round to nearest

5 Round to zero

6 Round up

7 Round down

When the modifier field is zero, rounding is controlled by the current rounding mode in the FPC register 210. When the field is not zero, rounding is performed as specified by the modifier, regardless of the current rounding mode. Rounding for modifiers 4–7 is the same as for rounding modes 0–3 (binary 00–11), respectively. Biased round to nearest (modifier 1) is the same as round to nearest (modifier 4), except when the second operand is exactly halfway between two integers, in which case the result for biased rounding is the next integer that is greater in magnitude.

A modifier other than 0, 1, or 4–7 is invalid. The sign of the result is the sign of the second operand, except that a zero result has a plus sign. Note that if the modifier is 5, the method of rounding is the same in the HFP and BFP modes.

Although the above description provides many specificities, these enabling details should not be construed as limiting the scope of the invention, and it will be readily understood by those persons skilled in the an that the present invention is susceptible to many modifications, adaptations, and equivalent implementations without departing from this scope and without diminishing its attendant advantages. It is therefore intended that the present invention is not limited to the disclosed embodiments but should be defined in accordance with the claims which follow.

What is claimed is:

1. A computer system, comprising:

a storage device including at least one stored bit for specifying one of plurality of rounding modes as a default rounding mode; and

a processor that executes a floating point instruction having a field that is operative to selectively override said default rounding mode with another of said rounding modes during execution of said floating point instruction by said processor;

wherein said rounding modes each specify a respective rounding direction applied to a floating point number.

2. The computer system according to claim 1, wherein said processor supports binary floating point format and hexadecimal floating point format.

3. The computer system according to claim 2, wherein said floating point instruction is common to said binary floating point format and said hexadecimal floating point format.

4. The computer system according to claim 1, wherein said floating point rounding modes include rounding modes specified by IEEE Std 754-1985 standards.

5. The computer system according to claim 1, wherein said floating point rounding modes include a biased round to nearest mode.

6. The computer system according to claim 1, wherein said floating point instruction includes a convert to fixed instruction.

7. The computer system according to claim 1, wherein said floating point instruction includes a load floating point integer instruction.

8. The computer system according to claim 1, wherein said field has a value which indicates that said default rounding mode is operative during execution of said floating point instruction by said processor.

9. A computer system including a plurality of floating point rounding modes, each of said floating point rounding modes specifying a respective rounding direction applied to a floating point number, said system comprising:

a storage element containing a value that specifies one of a plurality of rounding modes as a default rounding mode;

an instruction including a modifier field that selectively indicates another of said plurality of rounding modes; and

a processor that executes said instruction to provide a result of a given accuracy which is generated by rounding a floating point number having greater accuracy than the result, the rounding executed according to said default rounding mode when said modifier field does not override said default mode, and the rounding executed according to said another rounding mode when said modifier field overrides said default rounding mode, the rounding thereby being executed in response to said modifier field of said instruction.

10. The computer system according to claim 9, wherein said modifier field includes a value which indicates that said default rounding mode is operative during execution of said instruction by said processor.

11. A computer system including a plurality of floating point rounding modes, each of said floating point rounding modes specifying a respective rounding direction applied to a floating point number, said system comprising:

means for storing a value for specifying one of said floating point rounding modes as a default rounding mode; and

means for executing a floating point instruction having a field that is operative to selectively override said default rounding mode with another of said rounding modes during execution of said floating point instruction by said processor.

12. The computer system according to claim 11, wherein said field has a value which indicates that said default rounding mode is operative during execution of said floating point instruction.

* * * * *

EXHIBIT 19

US005825678A

United States Patent [19]

Smith

[11] Patent Number: 5,825,678

[45] Date of Patent: Oct. 20, 1998

[54] **METHOD AND APPARATUS FOR DETERMINING FLOATING POINT DATA CLASS**

[75] Inventor: Ronald M. Smith, Wappingers Falls, N.Y.

[73] Assignee: International Business Machines Corporation, Armonk, N.Y.

[21] Appl. No.: 414,858

[22] Filed: Mar. 31, 1995

[51] Int. Cl.[6] G06F 7/38; G06F 7/00; G06F 15/00; H03M 7/00

[52] U.S. Cl. 364/748; 364/715.03; 341/50; 341/94

[58] Field of Search 364/748, 715.03; 341/50, 89, 104, 105, 94; 395/375, 800

[56] References Cited

U.S. PATENT DOCUMENTS

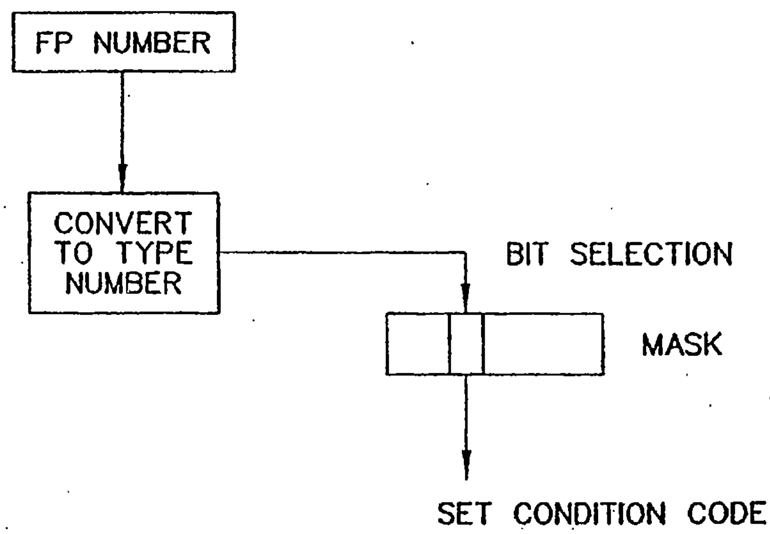
4,325,120 4/1982 Colley et al. 395/412
4,707,783 11/1987 Lee et al. 395/375
4,831,575 5/1989 Kuroda 364/748
5,191,335 3/1993 Leitherer 341/94

Primary Examiner—Reba I. Elmore
Assistant Examiner—Emmanuel L. Moise
Attorney, Agent, or Firm—Lynn Augspurger, Esq.; Morgan & Finnegan, LLP

[57] **ABSTRACT**

A new Test FP Data Class operation is provided which utilizes a 12-bit mask to determine to which of the 12 possible data classes a floating point number belongs and sets a condition code accordingly. As preferably embodied, a typical IBM System 390 instruction format is adapted to implement a Test FP Data Class operation. The class and sign of the first operand are examined to select one bit from the second-operand address. A condition code of 0 or 1 is set according to whether the selected bit is 0 or 1. The second-operand address is not used to address data; instead, individual bits of the address are used to specify the applicable combinations of operand calls and sign.

12 Claims, 6 Drawing Sheets



US005825678A

United States Patent [19]

Smith

[11] Patent Number: 5,825,678

[45] Date of Patent: Oct. 20, 1998

[54] METHOD AND APPARATUS FOR DETERMINING FLOATING POINT DATA CLASS

[75] Inventor: Ronald M. Smith, Wappingers Falls, N.Y.

[73] Assignee: International Business Machines Corporation, Armonk, N.Y.

[21] Appl. No.: 414,858

[22] Filed: Mar. 31, 1995

[51] Int. Cl.[6] G06F 7/38; G06F 7/00; G06F 15/00; H03M 7/00

[52] U.S. Cl. 364/748; 364/715.03; 341/50; 341/94

[58] Field of Search 364/748, 715.03; 341/50, 89, 104, 105, 94; 395/375, 800

[56] References Cited

U.S. PATENT DOCUMENTS

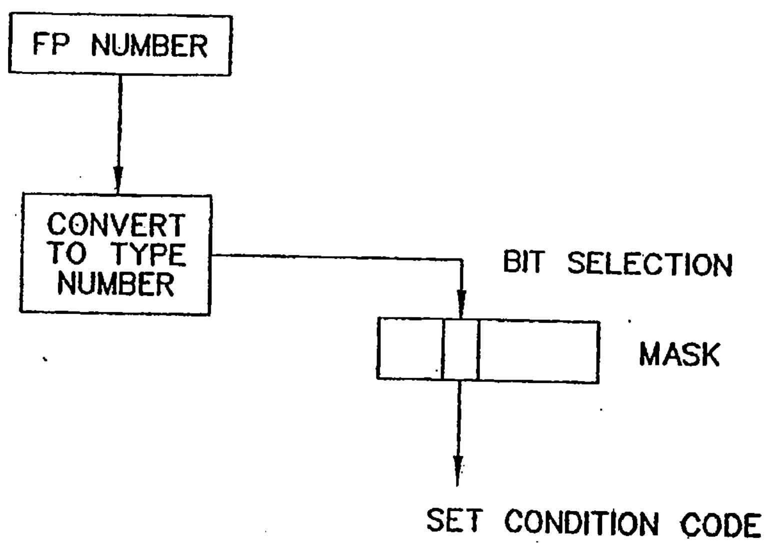
4,325,120	4/1982	Colley et al.	395/412
4,707,783	11/1987	Lee et al.	395/375
4,831,575	5/1989	Kuroda	364/748
5,191,335	3/1993	Leitherer	341/94

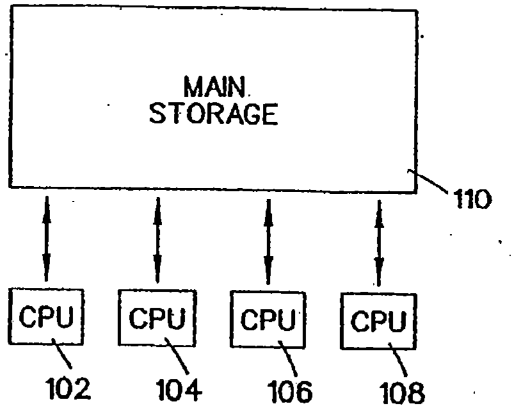
Primary Examiner—Reba I. Elmore
Assistant Examiner—Emmanuel L. Moise
Attorney, Agent, or Firm—Lynn Augspurger, Esq.; Morgan & Finnegan, LLP

[57] ABSTRACT

A new Test FP Data Class operation is provided which utilizes a 12-bit mask to determine to which of the 12 possible data classes a floating point number belongs and sets a condition code accordingly. As preferably embodied, a typical IBM System 390 instruction format is adapted to implement a Test FP Data Class operation. The class and sign of the first operand are examined to select one bit from the second-operand address. A condition code of 0 or 1 is set according to whether the selected bit is 0 or 1. The second-operand address is not used to address data; instead, individual bits of the address are used to specify the applicable combinations of operand calls and sign.

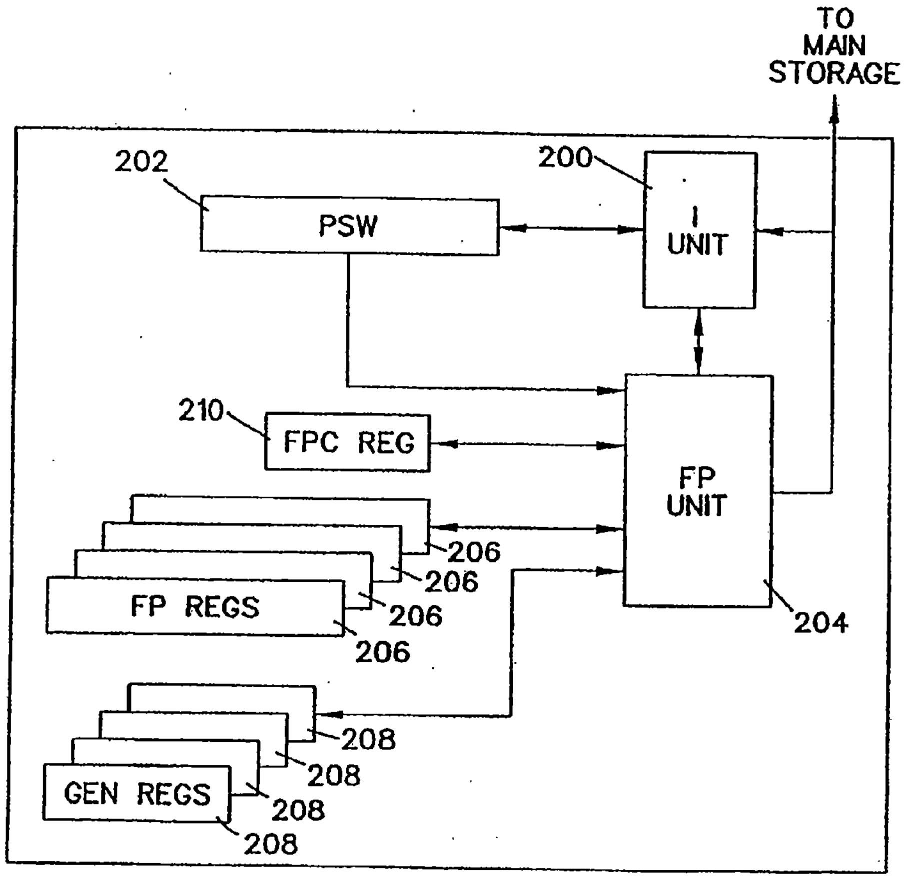
12 Claims, 6 Drawing Sheets





MAIN STORAGE
110
CPU
CPU
CPU
CPU
102
104
106
108

FIG. 1



TO MAIN STORAGE
202
200
PSW
I UNIT
210
FPC REG
FP UNIT
206
206
206
FP REGS
206
204
208
208
GEN REGS
208
208

FIG. 2

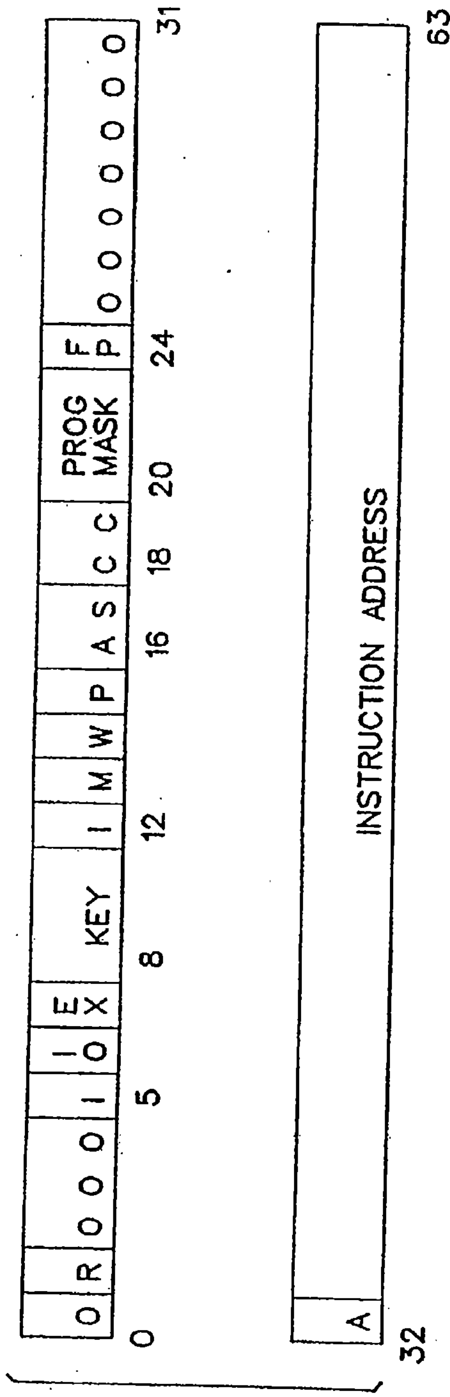

0
R
0
0
0
I
I O
E X
KEY
I
M
W
P
A S
C C
PROG MASK
F P
0
0
0
0
0
0
0
0
5
8
12
16
18
20
24
31
A
INSTRUCTION ADDRESS
32
63

FIG. 3

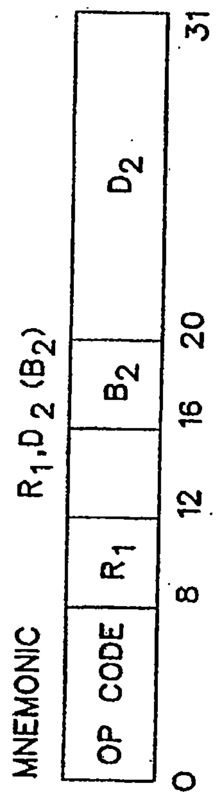

MNEMONIC
$R_1,D_2\ (B_2)$
OP CODE
R_1
B_2
D_2
0
8
12
16
20
31

FIG. 5

BYTE	BITS	VALUE	FUNCTION
0	0		BFP-INVALID-DATA MASK
	1		BFP-DIVISION-BY-ZERO MASK
	2		BFP-OVERFLOW MASK
	3		BFP-UNDERFLOW MASK
	4		BFP-INEXACT MASK
	5-7		(UNASSIGNED)
1	0		BFP-INVALID-DATA FLAG
	1		BFP-DIVISION-BY-ZERO FLAG
	2		BFP-OVERFLOW FLAG
	3		BFP-UNDERFLOW FLAG
	4		BFP-INEXACT FLAG
	5-7		(UNASSIGNED)
2	0-7		DATA-EXCEPTION CODE (DXC)
3	0		QNaN MODE
	1-5		(UNASSIGNED)
	6-7		ROUNDING MODE
		00	ROUND TO NEAREST
		01	ROUND TO ZERO
		10	ROUND UP
		11	ROUND DOWN

FIG.4

binary and hexadecimal floating point modes are supported, when the bit is zero, the CPU is in the hexadecimal-floating-point (HFP) mode, and floating-point operands are interpreted according to the HFP format. When the bit is one, the CPU is in the binary-floating-point (BFP) mode, and floating-point operands are assumed to be in BFP format. Some floating-point instructions operate the same in either mode.

When an instruction is executed which is not available in the current FP mode, a special-operation exception is recognized.

As illustrated in FIG. 4, the floating-point-control (FPC) register 210 is a 32-bit register, which contains the mode (i.e., rounding mode), mask, flag, and code bits. For this implementation, by way of example, the rounding mode is represented by the last two bits o f the last byte. Round to nearest, round to zero, round up, and round down modes are supported.

In FIG. 5 is illustrated a typical instruction format as is used, e.g., in an IBM System 390 computing system. As preferably embodied, this instruction format is adapted to implement the Test FP Data Class operation described herein. Further, three particular types of instructions for the Test FP Data Class operation may be implemented, i.e., mneumonics TCE, TCD, TCX corresponding to short, long, and extended operands, respectively. As adapted to the Test FP Data Class operation herein, R_1 designates the floating point register to be tested, and B_2 designates the base and D_2 is the offset used to form the second operand address. The class and sign of the first operand (R_1,) are examined to select one bit from the second-operand address ($D_2(B_2)$). Condition code 0 or 1 is set according to whether the selected bit is 0 or 1.

The second-operand address is not used to address data; instead, individual bits of the address are used to specify the applicable combinations of operand class and sign. In BFP mode, the rightmost 12 bits of the address, bits 20–31, are used to specify 12 combinations of operand class and sign; bits 0–19 of the second-operand class are ignored. Thus, as preferably embodied, B_2 may point to a general register which contains mask bits and D_2 would contain all zeros. Alternatively, when the B_2 field is all zeros, no base register is selected and D_2 may contain the mask bits. A third alternative where neither have all zeros is also possible, but not as practical.

FIG. 6 shows the definition of each of the six classes of BFP data including true zero, denormalized numbers, normalized numbers, infinity, quiet NaN, and signaling NaN. FIG. 7 illustrates the second operand address bits used for the Test FP Data Class operation in BFP mode.

In operation, a floating point processor conforming to IEEE standards provides an indication of the class of floating point number, which could be any one of the BFP data classes shown in FIG. 6. It may be appreciated that a program may want to know whether the floating point number is characterized by some combination of these BFP data classes. The Test FP Data Class operation provides the ability with one instruction to check all of the BFP data classes that the floating point number may fall into.

As illustrated in FIG. 8, a 12-bit mask is set, a bit corresponding to each of the possible classes and a particular bit is set to check for a particular data class. The floating point number in the floating point register that is pointed to by R_1 is loaded into the convert to type number logic and a determination is made as to the data class of the number. In the convert to type number logic, these floating point numbers are classified according to the sign, the fraction part, and the exponent part of the floating point number. That is, the floating point format has sign, exponent, and fraction bit. Based on these three bits, the floating point number may be categorized. Based upon the particular data class that it falls into, a signal is generated that will indicate one of the bits in the mask. That bit is then loaded into the condition code. That is, the condition code is set.

FIG. 9 illustrates the conversion circuitry of the convert to type number logic for bits 20 through 25 of the mask for the Test FP Data Class operation testing the short format (TCE). It will be appreciated that similar circuitry is used for bits 26 through 31 and for long (TCD) and extended (TCX) formats. In FIG. 9, the following abbreviations are used:

cc—condition code

en—exponent not zero and not all ones

eo—exponent all ones

ez—exponent zero

fn—fraction not zero

fz—fraction zero

sn—sign negative

sp—sign plus

By way of example, if it was to be determined whether the number is positive zero, bit location 20 in the mask field would be set to 1. Then, if the floating point number were a positive zero the convert to type number logic would cause the mask bit 1 to be loaded into the condition code storage location. If a positive zero is not to be tested, a 0 would be set in bit location 20 in the mask field and the 0 would be loaded into the condition code storage location.

In view of the foregoing, it may be appreciated that by simply checking one bit in the PSW, namely the condition code, any combination of the twelve data classes that the floating point number might fall into may be determined.

It may be further appreciated that more of the second-operand-address bits may be set to one. If the second-operand-address bit corresponding to the class and sign of the first operand is one, condition code 1 is set; otherwise, condition code 0 is set.

It may be further appreciated that operands, including signaling NaNs and quiet NaNs, may be examined without causing an arithmetic exception.

It may be further appreciated that the Test FP Data Class operation provides a way to test an operand without risk of an exception or setting BFP flags.

It may be further appreciated that the second-operand-address bits assigned for zero and normalized numbers are the same in the HFP and BFP modes. Thus, programs which use the Test FP Data Class operation to test only for these operand classes may be written in a mode-independent manner.

While the invention has been described in its preferred embodiments, it is to be understood that the words which have been used are words of description, rather than limitation, and that changes may be made within the purview of the appended claims without departing from the true scope and spirit of the invention in its broader aspects.

What is claimed:

1. An apparatus for determining floating point data class, comprising:

a floating point processor for interpreting a machine instruction to determine whether the data class of a floating point number is an identified data class;

means for retrieving the floating point number from memory;

means for determining whether the data class of the floating point number is the identified data class by examination of condition of the fields of the floating point number; and

means for setting a condition code in a program status word based upon the determination of whether the data class is the identified data class.

2. An apparatus for determining floating point data class in accordance with claim 1, wherein said means for determining uses a bit mask to determine action to be taken for a particular data class of the floating point number.

3. An apparatus for determining floating point data class in accordance with claim 2, wherein said machine instruction includes the location of said floating point number in memory and said bit mask.

4. An apparatus for determining floating point data class in accordance with claim 1, wherein said means for determining operates in a mode independent manner.

5. An apparatus for determining floating point data class in accordance with claim 1, wherein said machine instruction can accommodate floating point numbers having a short, long, or extended operand.

6. An apparatus according to claim 1, wherein said identified data class includes any specified combination of a plurality of possible data classes.

7. A method for determining floating point data class, comprising the steps of:

interpreting a machine instruction to determine whether the data class of a floating point number is an identified data class;

retrieving the floating point number from memory;

determining whether the data class of the floating point number is the identified data class by examination of condition of the fields of the floating point number; and

setting a condition code in a program status word based upon the determination of whether the data class is the identified data class.

8. A method for determining floating point data class in accordance with claim 7, wherein said step of determining further comprises the step of using a bit mask to determine action to be taken for a particular data class of the floating point number.

9. A method for determining floating point data class in accordance with claim 8, further comprising the step of utilizing a machine instruction including location of the floating point number in memory and the bit mask.

10. A method for determining floating point data class in accordance with claim 7, wherein said step of determining is accomplished in a mode independent manner.

11. A method for determining floating point data class in accordance with claim 7, further comprising the step of using a machine instruction accommodating floating point numbers having a short, long, or extended operand.

12. A method according to claim 7, wherein said identified data class includes any specified combination of a plurality of possible data classes.

* * * * *

EXHIBIT 20

US006654812B2

(12) **United States Patent**
Gioquindo et al.

(10) Patent No.: **US 6,654,812 B2**
(45) Date of Patent: **Nov. 25, 2003**

(54) **COMMUNICATION BETWEEN MULTIPLE PARTITIONS EMPLOYING HOST-NETWORK INTERFACE**

(75) Inventors: **Paul M. Gioquindo**, Poughkeepsie, NY (US); **Chin Lee**, Poughkeepsie, NY (US); **Bruce H. Ratcliff**, Red Hook, NY (US); **Stephen R. Valley**, Valatie, NY (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 78 days.

(21) Appl. No.: **09/977,799**

(22) Filed: **Oct. 15, 2001**

(65) **Prior Publication Data**

US 2002/0029286 A1 Mar. 7, 2002

Related U.S. Application Data

(63) Continuation of application No. 09/152,369, filed on Sep. 14, 1998, now Pat. No. 6,330,615, and a continuation of application No. 09/152,370, filed on Sep. 14, 1998, now Pat. No. 6,330,616, and a continuation of application No. 09/152,771, filed on Sep. 14, 1998.

(51) Int. Cl.^{7} **G06F 15/173**
(52) U.S. Cl. **709/236**; 370/466
(58) **Field of Search** 709/245–250, 709/228, 236, 218, 243, 252; 370/466, 467, 401, 406, 407

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,379,296 A	1/1995	Johnson et al.	370/60
5,560,038 A	9/1996	Haddock	709/236
5,596,723 A	1/1997	Romohr	709/222
5,623,605 A	4/1997	Keshav et al.	395/200.17

(List continued on next page.)

OTHER PUBLICATIONS

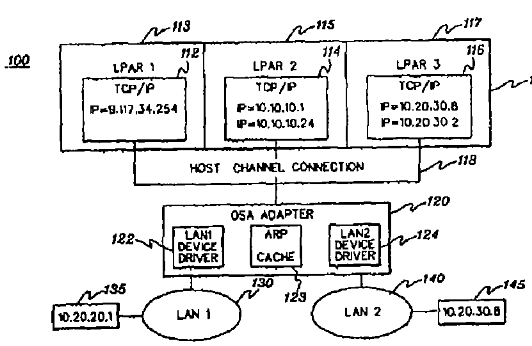
Plummer, D., RFC 0826, http://rfc.fh-koeln.de/rfc/html_gz/rfc0826.html.gz, 9 pages.
Hornig, C., RFC 0894, http://rfc.fh-koeln.de/rfc/html_gz/rfc0894.html.gz, 4 pages.
Postel, J., RFC 0895, http://rfc.fh-koeln.de/rfc/html_gz/rfc0895.html.gz, 4 pages.
Postel, J., RFC 1042, http://rfc.fh-koeln.de/rfc/html_gz/rfc1042.html.gz, 15 pages.
Renwick, J., RFC 1374, http://rfc.fh-koeln.de/rfc/html_gz/rfc1374.html.gz, 42 pages.
Garrett, J., RFC 1433, http://rfc.fh-koeln.de/rfc/html_gz/rfc1433.html.gz, 18 pages.
Perkins, C., RFC 2003, http://rfc.fh-koeln.de/rfc/html_gz/rfc2003.html.gz, 14 pages.
Comer, D., "Internet working with TCP/IP", vol. 1, Prentice-Hall, Inc., 3 pages.

Primary Examiner—B. Jaroenchoowanit
(74) *Attorney, Agent, or Firm*—Floyd A. Gonzalez, Esq.; Kevin P. Radigan, Esq.; Heslin Rothenberg Farley & Mesiti P.C.

(57) **ABSTRACT**

In a mainframe class data processing system having multiple logical partitions and a port to a network, a host-network interface is established for reducing network overhead at the multiple partitions. The host-network interface includes, for example, a host channel connection coupling the multiple partitions of the host system to a communications adapter having a network device driver for each network coupled to the adapter. The adapter also includes an address resolution protocol (ARP) cache designed to hold predetermined media headers for the clients coupled to the network(s) for use in forwarding an internet protocol (IP) datagram across the network to one of the clients from a partition of the host system. Provision is also made for partition-to-partition communication of IP datagrams by storing IP addresses of the logical partitions as HOME addresses in the ARP cache of the adapter.

15 Claims, 9 Drawing Sheets



U.S. PATENT DOCUMENTS

5,734,865 A	3/1998	Yu	395/500
5,740,438 A	4/1998	Ratcliff et al.	395/680
5,751,971 A	5/1998	Dobbins et al.	395/200.68
5,758,070 A	5/1998	Lawrence	709/200
5,835,720 A	11/1998	Nelson et al.	395/200.54
5,835,725 A	11/1998	Chiang et al.	395/200.58
5,850,526 A	12/1998	Chou	395/200.77
5,959,990 A	9/1999	Frantz et al.	370/392
6,016,388 A	1/2000	Dillon	395/200.72
6,018,767 A	1/2000	Fijolek et al.	709/218
6,032,197 A	2/2000	Birdwell et al.	709/247
6,047,325 A	4/2000	Jain et al.	709/227
6,049,826 A	4/2000	Beser	709/222
6,115,385 A	9/2000	Vig	370/401
6,128,294 A	10/2000	Oura et al.	370/389

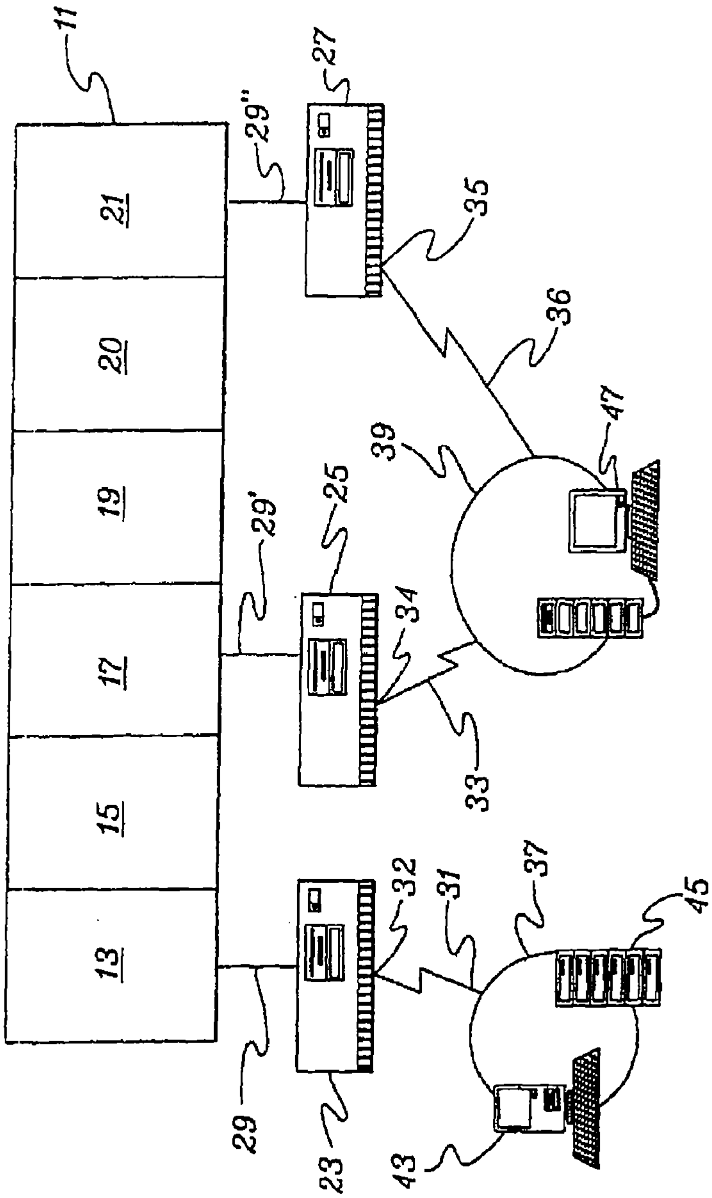


fig. 1
(PRIOR ART)

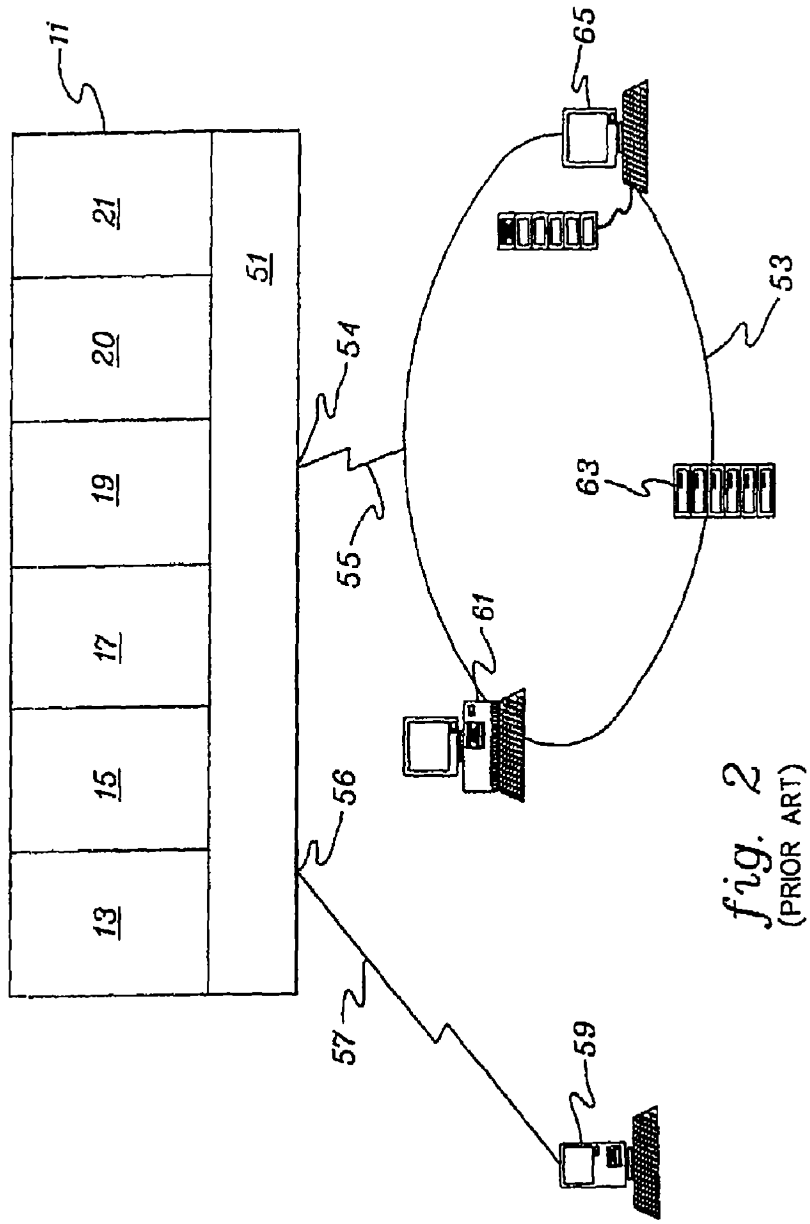
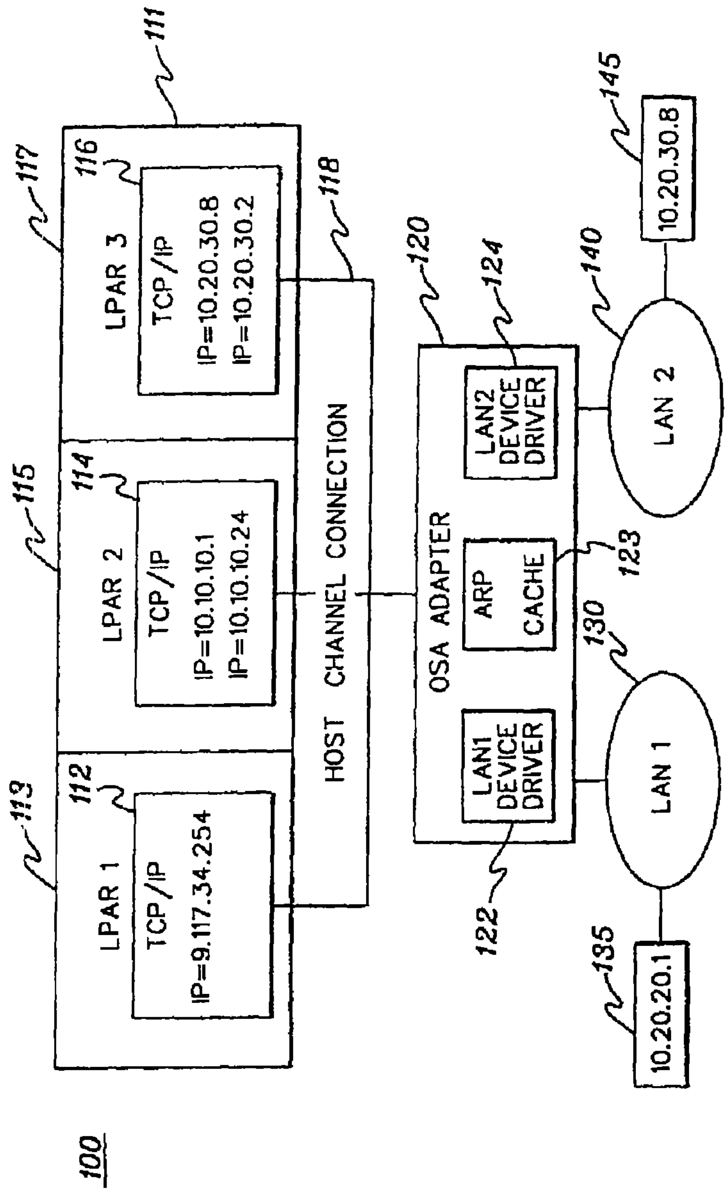


fig. 2
(PRIOR ART)



100
111
113
112
LPAR 1
TCP/IP
IP=9.117.34.254
115
114
LPAR 2
TCP/IP
IP=10.10.10.1
IP=10.10.10.24
117
116
LPAR 3
TCP/IP
IP=10.20.30.8
IP=10.20.30.2
118
HOST CHANNEL CONNECTION
120
OSA ADAPTER
122
LAN1 DEVICE DRIVER
ARP CACHE
123
124
LAN2 DEVICE DRIVER
130
LAN 1
135
10.20.20.1
140
LAN 2
145
10.20.30.8

fig. 3

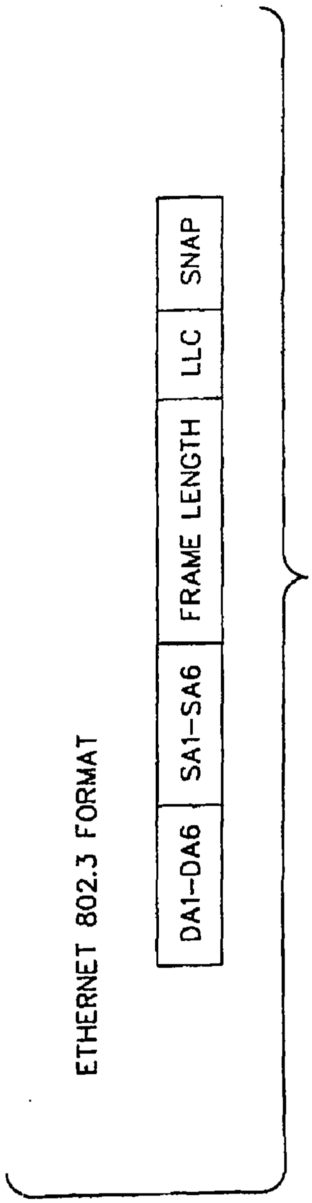

ETHERNET 802.3 FORMAT
DA1–DA6
SA1–SA6
FRAME LENGTH
LLC
SNAP

fig. 4A

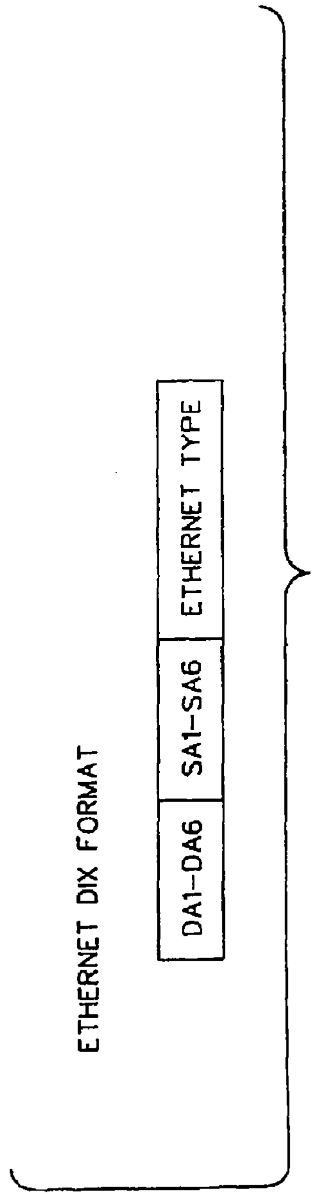

ETHERNET DIX FORMAT
DA1–DA6
SA1–SA6
ETHERNET TYPE

fig. 4B

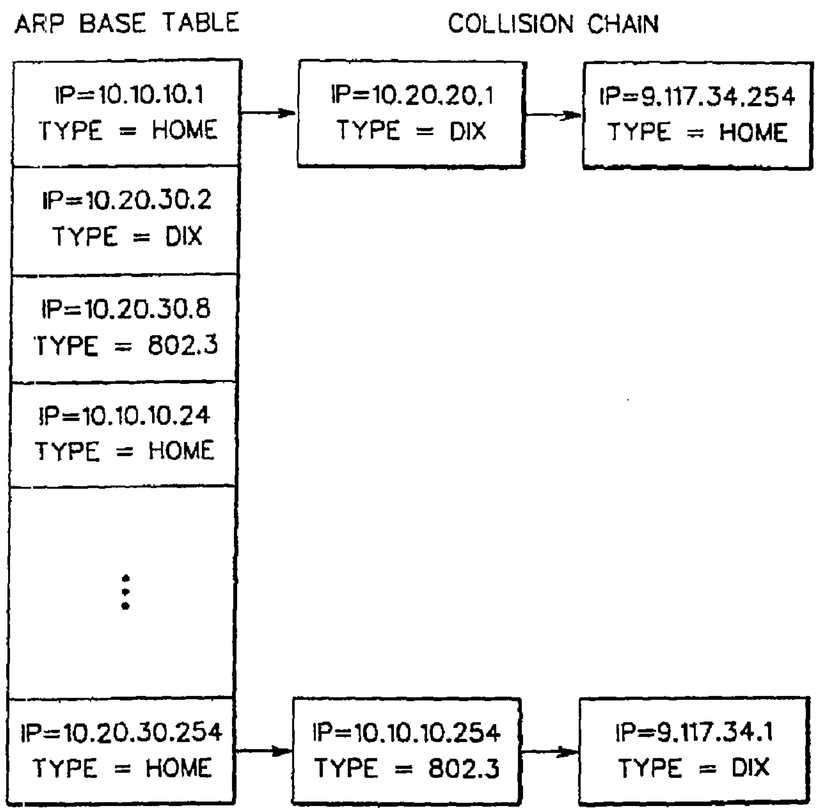

ARP BASE TABLE
COLLISION CHAIN
IP=10.10.10.1
TYPE = HOME
IP=10.20.20.1
TYPE = DIX
IP=9.117.34.254
TYPE = HOME
IP=10.20.30.2
TYPE = DIX
IP=10.20.30.8
TYPE = 802.3
IP=10.10.10.24
TYPE = HOME
IP=10.20.30.254
TYPE = HOME
IP=10.10.10.254
TYPE = 802.3
IP=9.117.34.1
TYPE = DIX

fig. 5

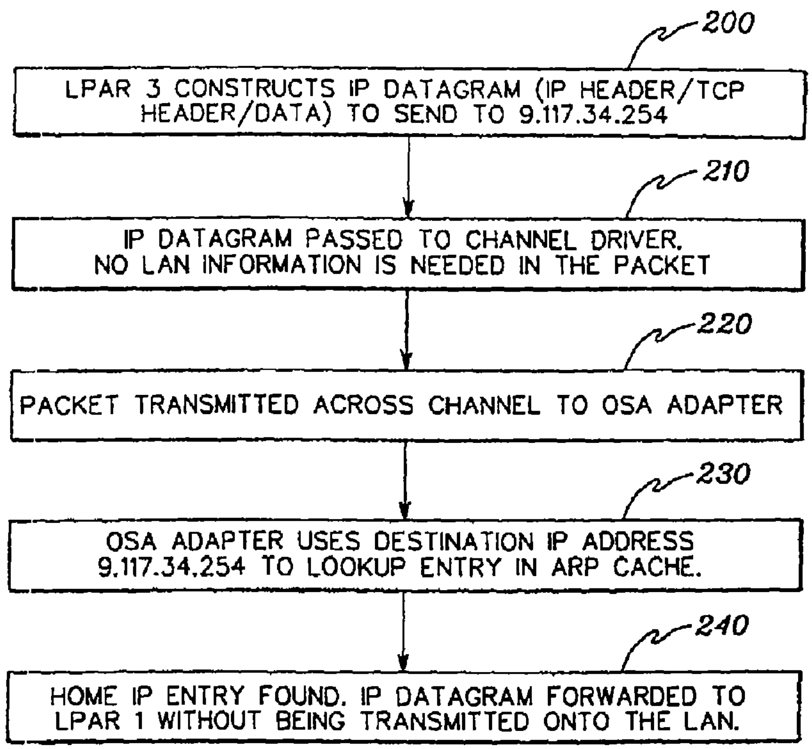

200
LPAR 3 CONSTRUCTS IP DATAGRAM (IP HEADER/TCP HEADER/DATA) TO SEND TO 9.117.34.254
210
IP DATAGRAM PASSED TO CHANNEL DRIVER. NO LAN INFORMATION IS NEEDED IN THE PACKET
220
PACKET TRANSMITTED ACROSS CHANNEL TO OSA ADAPTER
230
OSA ADAPTER USES DESTINATION IP ADDRESS 9.117.34.254 TO LOOKUP ENTRY IN ARP CACHE.
240
HOME IP ENTRY FOUND. IP DATAGRAM FORWARDED TO LPAR 1 WITHOUT BEING TRANSMITTED ONTO THE LAN.

fig. 6

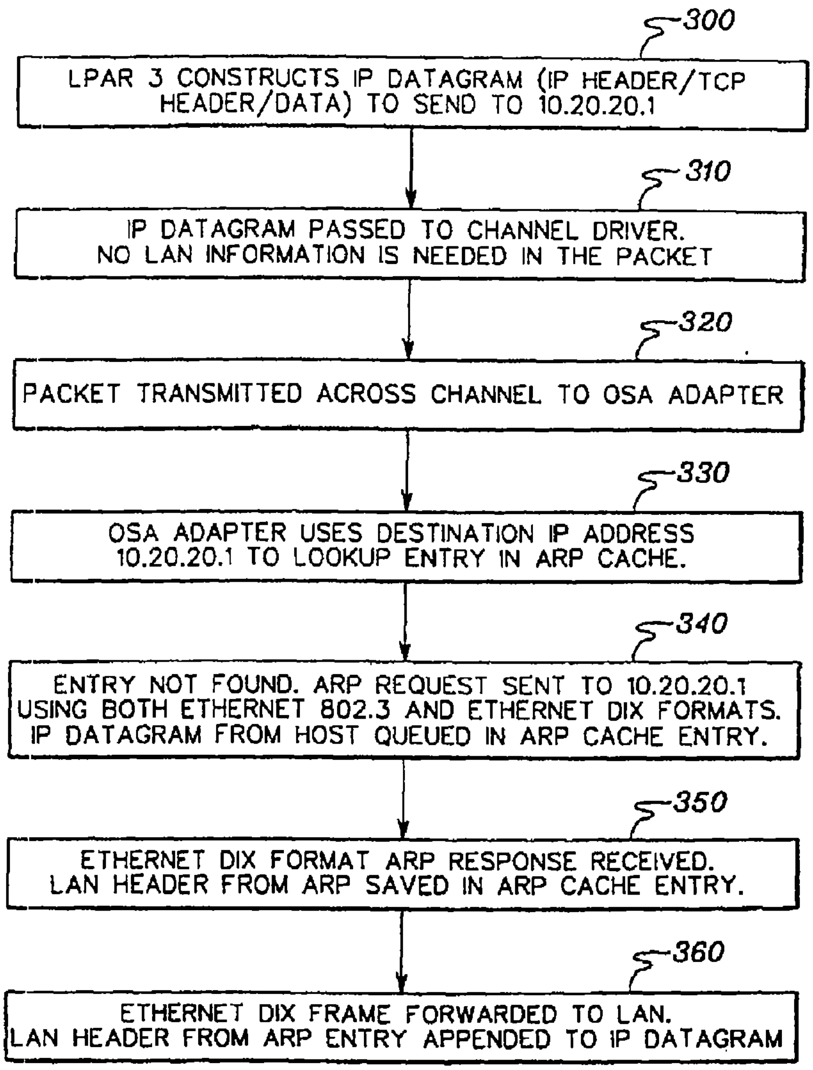

300
LPAR 3 CONSTRUCTS IP DATAGRAM (IP HEADER/TCP HEADER/DATA) TO SEND TO 10.20.20.1
310
IP DATAGRAM PASSED TO CHANNEL DRIVER. NO LAN INFORMATION IS NEEDED IN THE PACKET
320
PACKET TRANSMITTED ACROSS CHANNEL TO OSA ADAPTER
330
OSA ADAPTER USES DESTINATION IP ADDRESS 10.20.20.1 TO LOOKUP ENTRY IN ARP CACHE.
340
ENTRY NOT FOUND. ARP REQUEST SENT TO 10.20.20.1 USING BOTH ETHERNET 802.3 AND ETHERNET DIX FORMATS. IP DATAGRAM FROM HOST QUEUED IN ARP CACHE ENTRY.
350
ETHERNET DIX FORMAT ARP RESPONSE RECEIVED. LAN HEADER FROM ARP SAVED IN ARP CACHE ENTRY.
360
ETHERNET DIX FRAME FORWARDED TO LAN. LAN HEADER FROM ARP ENTRY APPENDED TO IP DATAGRAM

fig. 7

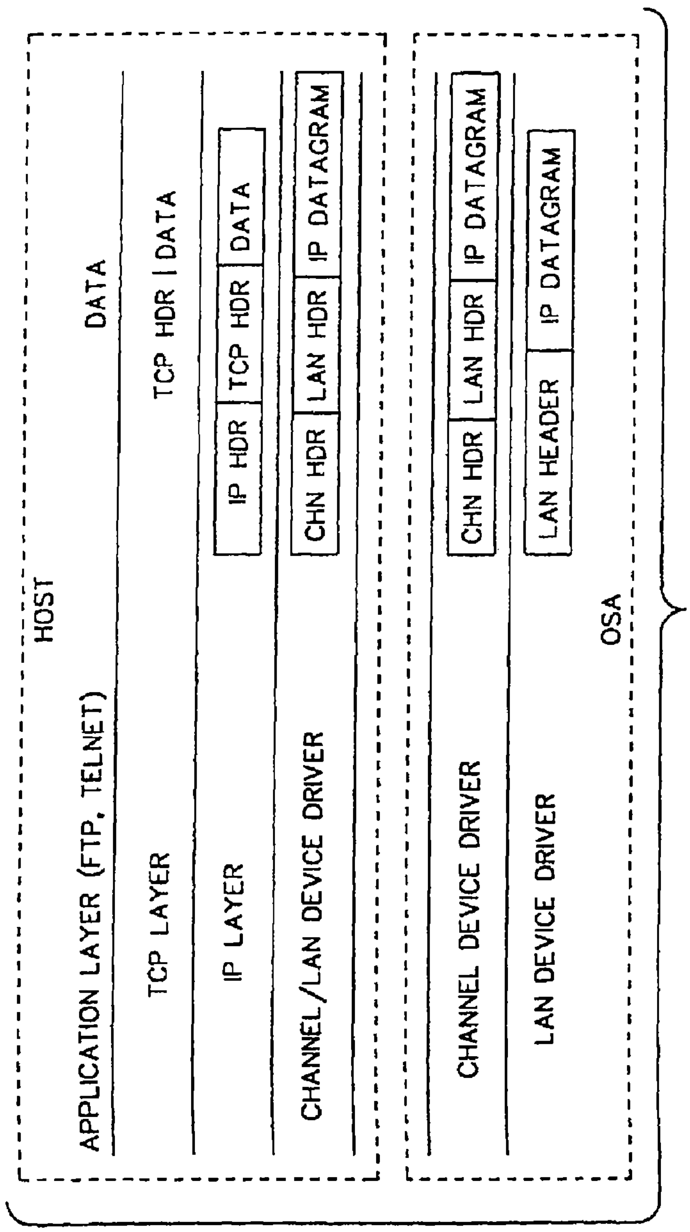

HOST
APPLICATION LAYER (FTP, TELNET)
DATA
TCP LAYER
TCP HDR | DATA
IP LAYER
IP HDR
TCP HDR
DATA
CHANNEL/LAN DEVICE DRIVER
CHN HDR
LAN HDR
IP DATAGRAM
OSA
CHANNEL DEVICE DRIVER
CHN HDR
LAN HDR
IP DATAGRAM
LAN DEVICE DRIVER
LAN HEADER
IP DATAGRAM

fig. 8A

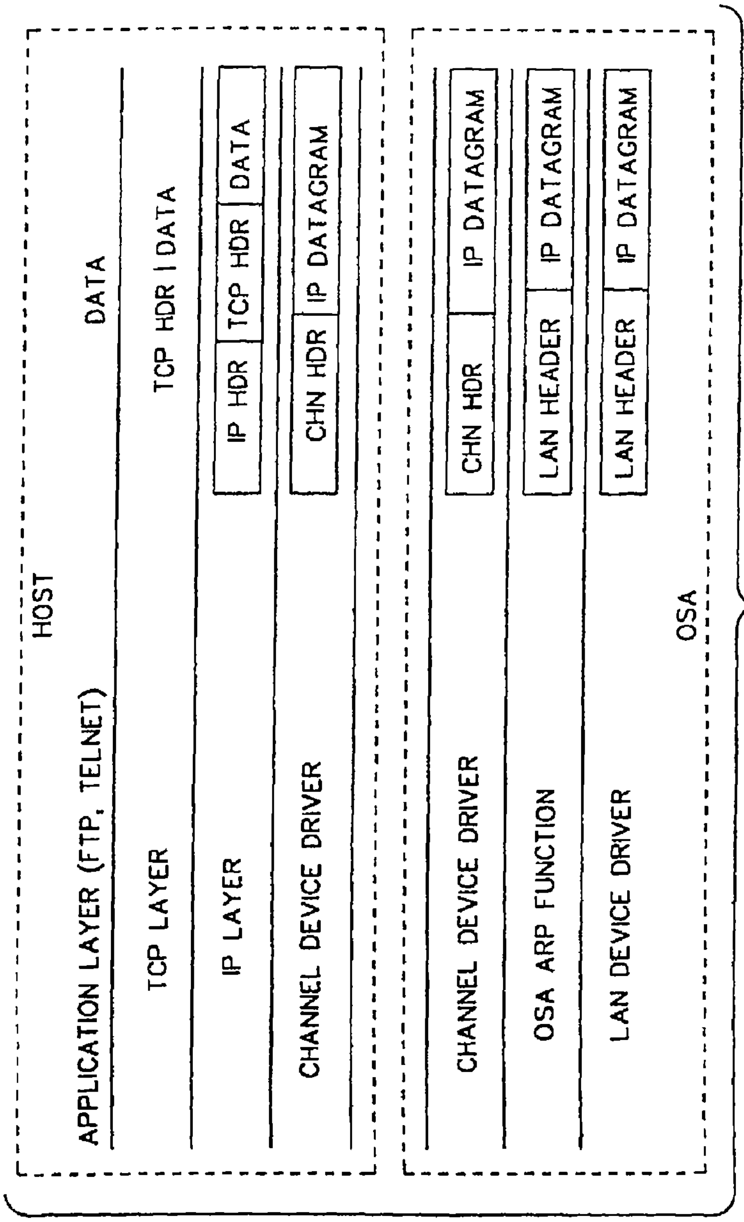

HOST
APPLICATION LAYER (FTP, TELNET)
DATA
TCP LAYER
TCP HDR I DATA
IP LAYER
IP HDR
TCP HDR
DATA
CHANNEL DEVICE DRIVER
CHN HDR
IP DATAGRAM
OSA
CHANNEL DEVICE DRIVER
CHN HDR
IP DATAGRAM
OSA ARP FUNCTION
LAN HEADER
IP DATAGRAM
LAN DEVICE DRIVER
LAN HEADER
IP DATAGRAM

fig. 8B

COMMUNICATION BETWEEN MULTIPLE PARTITIONS EMPLOYING HOST-NETWORK INTERFACE

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation application of and contains subject matter which is related to the subject matter of the following applications, each of which is assigned to the same assignee as this application. Each of the below-listed applications is hereby incorporated herein by reference in its entirety:

"METHOD FOR NETWORK COMMUNICATIONS OF MULTIPLE PARTITIONS EMPLOYING HOST-NETWORK INTERFACE", by Gioquindo et al., filed Sep. 14, 1998 now U.S. Pat. No. 6,330,615, Ser. No. 09/152,369;

"SYSTEM FOR NETWORK COMMUNICATIONS OF MULTIPLE PARTITIONS EMPLOYING HOST-NETWORK INTERFACE," by Gioquindo et al., filed Sep. 14, 1998 U.S. Pat. No. 6,330,616, Ser. No. 09/152,370; and

"NETWORK COMMUNICATIONS OF MULTIPLE PARTITIONS EMPLOYING HOST-NETWORK INTERFACE," by Gioquindo et al., filed Sep. 14, 1998, Ser. No. 09/152,771.

TECHNICAL FIELD

The present invention relates in general to network communications of processing systems. More particularly, the invention relates to techniques for effecting communications between a network and multiple partitions of a data processing system employing a host-network interface.

BACKGROUND OF THE INVENTION

Mainframe class data processing systems have hardware and software facilities that enable partitioning thereof. Such processing systems may be subdivided into multiple partitions whereby a user of a partition, or software executing in a partition, has the impression that the processing system is exclusively used by that application. Each partition has the appearance of being a separate and distinct processing system and may even run its own multi-tasking and multi-user operating systems independent from each other partition. An IBM Enterprise Systems Architecture ("ESA")/390 Mainframe Computer is an example of one such partitionable mainframe class data processing system. Partitioning thereof is described in, for example, various publications by International Business Machines Corporation, including "IBM ESA/390 Principles of Operation", IBM Publication No. SA22-7201-02, December 1994, and in the "IBM Enterprise System/9000 Processor Resource/Systems Manager Planning Guide", IBM Publication No. GA22-7123-11 (April 1994), which are both hereby incorporated herein by reference in their entirety.

Software executing in individual partitions within the mainframe class data processing system may require a network connection such as a local area network ("LAN") connection or a wide area network ("WAN") connection. This may be used to facilitate connectivity to users, or to application programs used in, for example, a client-server processing environment. Shown in FIG. 1 is a conventional configuration used to connect individual partitions, including the software running therein, to a LAN. The configuration includes a processing system 11 that has partitions 13, 15, 17, 19, 20 and 21.

Network connectivity for each partition of system 11 of FIG. 1 is achieved using separate network interfaces for each

partition. For example, partition 13 is conventionally connected through channel connection 29 to an IBM 3172 Interconnect Controller 23 (with 8232 Channel Interface Attachment) that has, for example, a token ring or Ethernet LAN port 32 attached to LAN 37 thereby providing LAN connection 31. Network connectivity is accordingly directly provided between partition 13 and computers 43 and 45 on LAN 37 through the IBM 3172 23. However, according to conventional techniques, this configuration has no other direct logical or physical connections from any of the other partitions to LAN 37. To further note, each application within partition 23 must communicate with a different network port on IBM 3172 23. The IBM 3172 (having internal 8232 Channel Interface Attachment), is described in a publication entitled "8232 LAN Channel Station", Apr. 15, 1998, IBM Publication No. ZZ25-8577-0, that is incorporated herein by reference in its entirety.

The conventional software executing on IBM 3172s restricts direct logical connectivity to being between a single partition and its corresponding LAN. Thus, to facilitate direct connectivity from a computer 47 on a LAN 39 to both partition 17 and 21, multiple IBM 3172s would traditionally be used. Partition 17 is coupled to LAN 35 via channel connection 29', IBM 3172 25 and LAN port 34 thereby establishing LAN connection 33. Similarly, partition 21 is coupled to LAN 39 via channel connection 29", IBM 3172 27, and LAN port 35 thereby establishing LAN connection 36.

The conventional host-to-network connectivity techniques discussed above have several limitations. Connectivity between a single network (i.e., LAN or WAN) and multiple partitions require the use of multiple interfaces therebetween such as, for example, multiple IBM 3172s. Further, each application executing in a single partition must use a different port on the IBM 3172 corresponding to the single partition.

An enhanced network interface for a mainframe class data processing system having multiple partitions and a port to a network is described in commonly assigned U.S. Pat. No. 5,740,438, which is hereby incorporated herein by reference in its entirety. Briefly summarized, this patent describes establishing a table which defines communications paths between the port to the network and at least two partitions of the multiple partitions. More specifically, each partition has at least one application executing therewithin and the communications paths are defined thereto. Data frames are passed between the network and the applications within the partitions through the port to the network and along the communications paths defined in the table such that the network communications is effected.

One embodiment of the network interface approach of U.S. Pat. No. 5,740,438 is depicted in FIG. 2. Shown is a partitionable mainframe class data processing system 11 (e.g., an IBM Enterprise System/9000) having an integral host-to-network interface ("HNI") 51 that facilitates a LAN connection 55 from multiple partitions 13, 15, 17, 19, 20 and 21 to LAN 53 through LAN port 54. Each application in each partition may directly communicate with computers 61, 63 and 65 on LAN 53 through the single host-to-network interface 51 and single LAN port 54. The LAN shown is a token ring LAN; however, the system is equally applicable to other types of LANs such as, for example, Ethernet and Fiber Distributed Data Interface ("FDDI"). Further, the host-to-network interface may support multiple network connections by way of multiple network ports. For example, a WAN connection 57 comprising, for example, a Peer-to-Peer Protocol ("PPP") connection may be established to a

computer 59 through WAN port 56. Any mix of LAN and WAN connections among multiple ports of host-to-network interface 51 is possible.

The host-to-network connectivity techniques described above have certain limitations, particularly in an Ethernet environment where two different frame types are possible, i.e., Ethernet DIX and Ethernet 802.3. For client/server systems, Transmission Control Protocol (TCP)/Internet Protocol (IP) has become the leading protocol for network communications. Using Ethernet, when a client application running over TCP/IP wants to communicate with a server application, the application must specify one of the two existing Ethernet frame formats. The frame format must also be known by the server machine in order for the TCP/IP connection to be established and any data transfer to occur. Conventionally, in order to make sure that the client and server communicate using the same Ethernet frame formats, both the client and server must specify the specific frame format to be used in their appropriate configuration files. If the configuration files do not match, then the client/server application will not work properly.

The most common server TCP/IP environment today has the complete TCP/IP functionality on one platform. For example, reference "TCP/IP Tutorial and Technical Overview," IBM Document No. GG24-3376-03 (December 1992). In this environment, one device driver exists in each partition for each LAN connection. Each device driver can specify a different Ethernet frame format, but will not support both frame formats. Thus, a different device driver is used for each of the two frame formats. In operation, a dedicated device driver of a partition of the host system takes care of providing both channel headers and media or LAN headers necessary for transmission of an IP packet across the LAN to a client coupled thereto. Again, however, this dedicated device driver is configured to function with one particular type of LAN, and with respect to Ethernet, one particular frame format.

With the above as background, the present invention is directed to enhancements to the state-of-the-art of network communications of multiple partitions employing a host-network interface.

DISCLOSURE OF THE INVENTION

Briefly summarized, the present invention comprises in one aspect a method of network communications implemented within a host-network interface for use in a mainframe class data processing system having multiple partitions, and a port to a network. This method includes: saving at the host-network interface an internet protocol (IP) address for at least one of the multiple partitions of the mainframe class data processing system; generating an IP datagram at a first partition of the multiple partitions to be forwarded to a second partition of the multiple partitions using a destination IP address; and determining whether the destination IP address for the IP datagram comprises a saved IP address at the host-network interface of the at least one partition of the mainframe class data processing system, and if so, forwarding the IP datagram directly from the first partition to the second partition without employing the network.

Systems and computer program products corresponding to the above-summarized method are also described and claimed herein.

Network communications in accordance with the principles of the present invention provides numerous advantages over the existing art for a mainframe class data processing system having a port coupled to a network. For example, reduced configuration information is required at the host system since the user application does not have to specify the network type in the host configuration. In one aspect, this is accomplished by providing a technique for dynamically determining an Ethernet frame format at the communications adapter of the host-network interface coupling the host system to the network. Further, in accordance with this invention, only one common non-network specific channel device driver is needed at each logical partition of the host system, and one network, e.g., LAN, device driver at each port of the network adapter. Less internet protocol addresses are therefore needed for host connections to the network. Advantageously, one single IP address within a partition can be used to communicate with any number of different networks, e.g., token ring, Ethernet DIX or Ethernet 802.3. This contrasts with existing configurations, wherein for each unique network type, and for each Ethernet frame type, a different IP address must be specified within the partition. Further, partition-to-partition traffic is facilitated herein without requiring a network connection to route an internet protocol (IP) datagram from a first logical partition to a second logical partition of the host data processing system.

BRIEF DESCRIPTION OF THE DRAWINGS

The above-described objects, advantages and features of the present invention, as well as others, will be more readily understood from the following detailed description of certain preferred embodiments of the invention, when considered in conjunction with the accompanying drawings in which:

FIG. 1 is a system diagram of a conventional network connected partitionable mainframe class data processing system;

FIG. 2 is a system diagram of an alternate embodiment of a network connected partitionable mainframe class data processing system;

FIG. 3 is a system diagram of one embodiment of a network connected partitionable mainframe class data processing system pursuant to the present invention;

FIGS. 4a & 4b are simplified block diagrams of the two Ethernet frame formats in common use;

FIG. 5 depicts one embodiment of an Address Resolution Protocol (ARP) cache implemented within a host-network interface in accordance with the present invention;

FIG. 6 is a flow diagram of communications in accordance with the present invention between a first partition and a second partition in a common mainframe class data processing system;

FIG. 7 is a flow diagram of communications in accordance with the present invention between a partition of the system and a client on a network coupled to the mainframe class data processing system;

FIG. 8a is a block diagram of a conventional embodiment of the layers to be constructed for an internet protocol (IP) packet to be transmitted from a partition at the host system onto a local area network (LAN) via an adapter coupled between the host system and the network; and

FIG. 8b is a block diagram of one embodiment of the layers to be constructed for an IP packet to be transmitted from the host system, across the adapter and onto a LAN in accordance with the present invention.

BEST MODE FOR CARRYING OUT THE INVENTION

As noted, a common communications protocol for the mainframe environment today is the Transmission Control

Protocol/Internet Protocol (TCP/IP). TCP/IP communications is described in, for example, the above-noted document entitled "TCP/IP Tutorial and Technical Overview," IBM Document No. GG24-3376-03, published December, 1992, which is hereby incorporated herein by reference in its entirety. The present invention employs, e.g., TCP/IP communications with certain TCP/IP functionality being migrated from the host system to a second platform, herein referred to as the host-network interface. FIG. 3 depicts one embodiment of a communications network 100 in accordance with the present invention wherein an IBM S/390 host system 111 is coupled to multiple local area networks across a host-network interface which comprises a host channel connection 118 and a communications adapter, such as an IBM Open Systems Adapter (OSA) 120. The OSA 120 may be platform channel attached to or integrated with the mainframe environment. In the OSA environment, the TCP/IP address resolution protocol (ARP) function has been migrated to the OSA platform 120.

In accordance with the present invention, the OSA adapter has the capability of supporting any number of different TCP/IP connections through a common network device driver, such as LAN 1 device driver 122 and LAN 2 device driver 124. Further, pursuant to this invention, the HOST applications are no longer required to know anything about the LAN or wide area network (WAN) media over which they are communicating. Since only one channel device driver per network is to be employed, provision is made herein for the network device driver to support both Ethernet DIX and Ethernet 802.3 formats. In addition, the OSA has the capability to dynamically determine which Ethernet frame type a particular client platform supports for a current connection. Each of these aspects of the invention, as well as others, is described in detail hereinbelow.

FIG. 3 shows three separate HOST logical partitions (LPARs) or partitions. As is well known, each LPAR functions as its own separate entity, with each logical partition (LPAR) of the host system 111 being viewed as a completely separate entity. The TCP/IP box within each LPAR defines a unique TCP/IP stack 112, 114, 116, which can function independently of the others. The IP fields are the unique IP addresses defined for each TCP/IP stack. A TCP/IP stack can have one or more IP addresses defined.

Each logical partition LPAR 1 113, LPAR 2 115, LPAR 3 117, couples through a Host Channel Connection 118 to a communications adapter 120, such as IBM's Open Systems Adapter (OSA). The OSA adapter, which has a connection to each of the TCP/IP stacks, is an integrated channel attached LAN adapter which does not contain its own TCP/IP stack. The Host Channel Connection 118 comprises as one example an IBM Enterprise System Connection ("ESCON") channel connector. The LAN device drivers 122, 124 and the Address Resolution Protocol cache 123 are contained in the OSA adapter 120. This is different from other channel attached adapters which do not contain the IP functionality. For background on an Address Resolution Protocol (ARP) cache or table, reference a textbook by W. Richard Stevens entitled *TCP/IP Illustrated*, published by Addison-Wesley, Volume 1 (1994), the entirety of which is hereby incorporated herein by reference.

Previously, each logical partition was required to contain its own LAN device driver. Pursuant to the present invention, however, multiple logical partitions are shown in FIG. 3 as sharing a common network device driver 122 or 124 for accessing network 130, 140, respectively, coupled to the host system 111. By way of example, OSA adapter 120 is shown connected to two different LANs, i.e., LAN 1 130 and LAN 2 140. These LANs can be different media types, e.g., Fiber Distributed Data Interface ("FDDI"), token ring, or Ethernet. Further, each LAN 130, 140 includes a representative client platform 135 & 145. The numbers depicted in the client platforms are representative of IP addresses used in accessing the particular platforms.

As briefly noted above, one aspect of this invention comprises providing the host-network interface with the capability to dynamically determine or predetermine the particular Ethernet frame format needed for a LAN device driver, for example, LAN 1 device driver 122, to communicate across the network with a client 135 coupled to the network. In an Ethernet environment, there are two different frame formats in common use, i.e., Ethernet 802.3 and Ethernet DIX, for client platforms which can co-exist on the Ethernet network. FIGS. 4a & 4b depict the fields of the Ethernet 802.3 and Ethernet DIX formats, respectively. A description of these fields is set forth below:

- DA1–DA6—this field is the destination MAC address which comprises the Media Access Control address of a port on the destination LAN adapter. By way of example, the destination address is six bytes in length.
- SA1–SA6—this is the source MAC address which comprises the media access control address of a port on the sending LAN adapter. In this example, the source address is six bytes.
- Frame length—this is the length of an Ethernet frame, including length of LLC, SNAP and IP datagram. In one example, the frame length is two bytes.
- LLC—this field comprises the Logical Link Control field and contains, for example, protocol information. Protocol is the networking protocol that the application is using. Examples of networking protocols include TCP/IP, "NETBIOS," "IPX" (used in "Novell" networks) and Systems Network Architecture ("SNA"). The LLC field in Ethernet 802.3 format may comprise three bytes.
- SNAP—identifies the Sub-Network Access Protocol within the LAN packet. In one example, the SNAP header may be five bytes.
- Ethernet Type—identifies the protocol of the data that follows the Ethernet type in the frame. The Ethernet type of Ethernet DIX format is two bytes in length.
- MAC—this is the Media Access Control field which contains, for example, broadcast indications and other LAN specific information.

To eliminate from the host the conventional configuration information otherwise needed for each IP session to identify the Ethernet frame type used between a client/server application, the present invention implements a technique which dynamically solicits the client platform to determine which Ethernet frame type is being used. To accomplish this, the OSA adapter implements the following:

- When an IP datagram is received from a host partition which is to be routed to a new Ethernet client platform, two separate ARP packets are constructed. A first ARP packet is constructed using the Ethernet DIX frame format and a second ARP packet is constructed using the Ethernet 802.3 format. Both frame formats are then broadcast across the network.
- The client platform only responds to the ARP request which has the proper frame format. The other ARP will be dropped by the destination client platform upon its receipt.
- When the client's ARP response is received back at the source system, its OSA adapter will know which frame

format to use for the pending or future IP session(s) with that client.

Advantageously, employing the above concept results in less configuration information being needed at the host system. That is, the user application does not have to specify the LAN type in its host configuration.

Along with dynamically identifying the Ethernet frame format being used by a client machine, the present invention also includes storing media headers in the ARP table (or ARP cache) employed by the communications adapter. (As used herein, a "media header" refers to the MAC header, and is also referred to as a LAN header or WAN header.) For example, upon receipt of an ARP response from a client to a broadcast ARP request, the LAN header which defines the Ethernet frame format of the client machine is obtained from the response and stored into the ARP table in a manner which will be straightforward to one of ordinary skill in the art. The ARP table will thus contain two headers, one for ARP packets and one for IP packets. For Ethernet 802.3 headers, the entire MAC/LLC/SNAP header is stored in the ARP table. For Ethernet DIX, only the MAC header is needed. For a client which supports both Ethernet frame formats, the frame format in the first response from the client received at the OSA adapter may be used.

By storing the media header in the ARP table, the OSA adapter can then subsequently append this header to every IP datagram being sent to that particular client platform without having to construct a new media header each time. For Ethernet DIX packets, all the fields in the LAN header are constant, while for Ethernet 802.3, all the fields are constant except for the Frame Length field. This field must be changed to the packet length for each packet transmitted.

Further, media headers can be predetermined by configuring the communications adapter to monitor ARP requests received from the network. Each incoming request will also identify the Ethernet frame type being used by the source client. From such a request, the LAN headers can be obtained and pre-stored in the ARP cache for future use with an IP request destined for that client platform as described above. Advantageously, by so pre-storing the LAN headers, the header does not have to be constructed for each IP datagram being transmitted from the adapter and the host system is freed of any responsibility for designating a client's Ethernet frame format.

Another aspect of the present invention comprises designating and saving "HOME" IP addresses within the communications adapter, e.g., within the ARP cache. FIG. 5 presents an example of the possible formats of the ARP entries on an OSA adapter in accordance with one embodiment of the invention. The figure depicts two tables, namely, an ARP Base Table and an ARP Collision Chain. To find a proper entry in the OSA ARP table, an IP address is traditionally first hashed to compute a one byte index into the ARP Base Table, which may by way of example comprise 256 bytes. Since it is possible for two IP addresses to "hash" to the same index, the ARP Collision Table is needed to chain these additional entries to the ARP Base Table. The IP numbers in the two tables comprise examples of IP addresses for the application used by the particular networking protocol as will be apparent to one of ordinary skill in the art. The examples agree in part with the IP addresses provided in FIG. 3.

The information in the ARP tables is preferably obtained from the TCP/IP stacks at initialization time and from the LAN as ARP requests/responses received from the network. At initialization time, each Host TCP/IP stack is configured to register its HOME IP addresses with the OSA adapter in a manner apparent to one skilled in the art. The "HOME" addresses are those which are recognized as local IP addresses by the specific stack. These entities are marked as HOME entries in the ARP cache. The entries are unique from the ARP entries for the LAN because they do not contain the specific LAN frame type which is to be used in transmitting a packet. If needed, the frame format for these entries is dynamically determined as described above by the client or server in the network which sends packets to the OSA adapter.

The HOME IP addresses mark the IP addresses to which OSA will respond when receiving an ARP request from the LAN. The format of the ARP response which OSA returns is the same as the format of the ARP request which OSA receives. The entire MAC header used to send back an ARP response is thus also saved in the ARP tables on the OSA adapter. This MAC header is then appended to any IP datagrams received from the Host which are destined for the matching IP address.

The HOME entries serve another purpose. In accordance with a further aspect of the invention, these entries are preferably used to route packets from a first logical partition (LPAR) to a second LPAR of the host system without going onto a network. When a packet is received from the Host, the destination IP address is "looked up" in the ARP tables of the host-network interface. If an entry is found and it is marked as a HOME entry, then the IP packet is routed directly to the LPAR owning that address. Since the packet is not sent out onto a network, no media or network header needs to be constructed.

FIG. 6 depicts one embodiment of LPAR to LPAR communications in accordance with this aspect of the invention. In this example, a logical partition, LPAR 3, is assumed to construct an IP datagram which includes an IP header, a TCP header and data, for sending to an IP address 200, which is assumed to comprise address 9.117.34.254 (see FIGS. 3 & 5). The IP datagram is then passed to, e.g., an IBM ESCON channel driver within host channel connection 118 (FIG. 3) of the host-network interface, and no LAN information is provided by the LPAR in the packet 210. The packet is transmitted across the channel to, e.g., an OSA adapter 220 and the OSA adapter uses the destination IP address 9.117.34.254 to look up the entry in the ARP cache 230. As shown in FIG. 5, this particular IP address is designated in the collision chain as a HOME type IP address. Since a HOME IP entry is found, the IP datagram is forwarded directly to LPAR 1 from LPAR 3 without being transmitted out onto a network 240.

Packets for which a LAN entry is found are necessarily sent onto the appropriate LAN. In this example, a "LAN entry" comprises a saved IP address with a format type of Ethernet DIX or Ethernet 802.3 (see FIG. 5). For Ethernet DIX packets, the exact media or MAC header stored in the ARP table as described above is appended to the IP datagram. The packet is then forwarded to the respective LAN driver (FIG. 3) and sent on to the LAN. For Ethernet 802.3 packets, the LAN header stored in the ARP table is copied into a unique buffer area. The frame length field in the LAN header is then modified to reflect the LLC/SNAP/IP datagram total length, and the header is appended to the IP datagram and forwarded to the LAN driver.

Packets received for which an entry cannot be found in the ARP table are preferably queued in the ARP table. As noted above, ARP requests in both Ethernet 802.3 and Ethernet DIX formats are then transmitted onto the LAN, i.e., assuming that the LAN comprises one of the two Ethernet formats. Any subsequent IP datagrams received from the Host prior

to an ARP response being returned from the client across the LAN are also preferably queued from the ARP table entry which is "pending" the ARP response. Once an ARP response is received at the OSA adapter, all the queued packets are transmitted onto the LAN.

FIG. 7 depicts one embodiment of LPAR to LAN communication in accordance with this aspect of the invention. A logical partition, for example, LPAR 3, constructs an IP datagram which includes an IP header, TCP header and data, for sending to an IP address, designated 10.20.20.1 300. This particular address is depicted in FIG. 3 as comprising a client coupled to LAN 1 and in FIG. 5 as being Ethernet DIX frame format.

The IP datagram is passed to the channel driver with no LAN information in the packet 310, for forwarding to the OSA adapter 320. The OSA adapter uses the destination IP address 10.20.20.1 to look up the entry in the ARP cache 330. Assuming that this is the first request to that particular address, then no entry will be found. An ARP request thus is sent to IP address 10.20.20.1 using both Ethernet 802.3 and Ethernet DIX formats. IP datagrams from the Host are queued in the ARP cache entry awaiting a response from this IP address 340. An Ethernet DIX format ARP response is assumed received and the LAN header from the ARP response is saved in the ARP cache entry 350. For example, reference FIG. 5 where in the example shown the IP address, and LAN or media header, is placed into the collision chain in a manner that will be apparent to one of ordinary skill in the art. The Ethernet DIX framed IP datagram is then forwarded to the LAN, with the LAN header from the ARP entry appended to the IP datagram received from the logical partition 360.

To summarize, by having the ARP code and the device drivers present on the OSA adapter, the Host's TCP/IP stacks are given the ability to define one IP address only and have it communicate to any LAN type through the adapter. Prior to this invention, the host system had to have one unique LAN device driver for each LAN type. Further, for Ethernet formats, one LAN device driver was needed to be loaded for Ethernet DIX connections and another for Ethernet 802.3 connections. By placing all the LAN specific drivers in the OSA adapter along with the ARP function, one IP connection can communicate to any other IP connection either LPAR to LPAR or LPAR to LAN using one generic channel device driver.

FIG. 8a shows the various layers involved in the constructing of a packet to be transmitted onto a LAN pursuant to conventional processing wherein the host system employs a unique LAN device driver for each LAN type to which the host is connected. The HOST contains the application, TCP, IP and channel/LAN device driver layers. The flow starts at the application layer. For example, a user issues a file transfer program (FTP) command to send a file to a client. The FTP (application) then passes the data to the TCP layer. The TCP encapsulates the FTP data with a TCP header. The TCP packet is then passed to the IP layer which encapsulates the TCP packet with an IP header. The IP header/TCP header/user data packet is known as the IP datagram. The IP datagram is then passed to the channel/LAN device driver where the IP datagram is encapsulated with a LAN header and a channel header to send the packet across, for example, an ESCON channel to an OSA adapter. In order for the LAN header to be injected at the host, the host conventionally requires knowledge of the LAN.

Once the packet is received by OSA, the channel header is stripped and the packet is passed to the LAN device driver which transmits the packet onto the proper LAN media.

For comparison, FIG. 8b shows the various layers involved in constructing a packet to be transmitted on the LAN in accordance with the present invention. The HOST contains the application, TCP, IP and channel device driver layers. The flow starts at the application layer. For example, a user issues an FTP command to send a file to a client. The FTP (application) then passes the data to the TCP layer. TCP encapsulates the FTP data with a TCP header. The TCP packet is then passed to the IP layer which encapsulates the TCP packet with an IP header. The IP header/TCP header/user data packet is known as the IP datagram. The IP datagram is then passed to the channel device driver. The channel device driver encapsulates the IP datagram with a channel header to send the packet across the ESCON channel to the OSA adapter.

Once the packet is received by OSA, the channel header is stripped and the packet is passed to the OSA ARP function. The OSA ARP function appends the LAN header from the associated ARP cache entry to the IP datagram and passes the packet to the LAN driver. Since the ARP function constructed the entry's LAN header, the LAN driver does not need to construct a header, it just transmits the packet onto the proper LAN media.

One skilled in the art will recognize from a comparison of FIGS. 8a & 8b that the present invention relieves the host partitions of processing overhead associated with knowing the LAN configuration and moves the responsibility for providing the media header to the communications adapter.

As one further consideration, for IP networks which run in a LAN environment, the IP datagram which is transmitted must not exceed the MTU (Maximum Transmission Unit). The MTU is determined from the LAN media on which the IP datagram is being transmitted. For example, for Ethernet LANs, the maximum MTU is 1500 bytes for Ethernet DIX frames and 1492 for Ethernet 802.3 frames. Since the MTU size determines the maximum size for an IP datagram, the smaller the MTU, the more IP datagrams which must be constructed to transmit a block of data across a network. This has a direct effect on the performance of an IP connection. For LPAR to LPAR traffic, in accordance with the present invention, the IP datagrams are not sent onto the LAN environment. Therefore, for LPAR to LPAR connections, the Host configuration can specify a very large MTU (currently 64 K) which will dramatically increase the performance of the IP connection.

The present invention can be included, for example, in an article of manufacture (e.g., one or more computer program products) having, for instance, computer usable media. This media has embodied therein, for instance, computer readable program code means for providing and facilitating the capabilities of the present invention. The articles of manufacture can be included as part of the computer system or sold separately. Additionally, at least one program storage device readable by machine, tangibly embodying at least one program of instructions executable by the machine, to perform the capabilities of the present invention, can be provided.

The flow diagrams depicted herein are provided by way of example. There may be variations to these diagrams or the steps (or operations) described herein without departing from the spirit of the invention. For instance, in certain cases, the steps may be performed in differing order, or steps may be added, deleted or modified. All of these variations are considered to comprise part of the present invention as recited in the appended claims.

While the invention has been described in detail herein in accordance with certain preferred embodiments thereof,

many modifications and changes therein may be effected by those skilled in the art. Accordingly, it is intended by the appended claims to cover all such modifications and changes as fall within the true spirit and scope of the invention.

What is claimed is:

1. A method of network communications implemented within a host-network interface for use in a mainframe class data processing system having multiple partitions and a port to a network, said method comprising:

saving at said host-network interface an internet protocol (IP) address of at least one of said multiple partitions of the mainframe class data processing system;

generating an IP datagram at a first partition of said multiple partitions to be forwarded to a second partition of said multiple partitions using a destination IP address; and

determining whether said destination IP address for said IP datagram comprises an IP address saved at said host-network interface for said at least one partition, and if so, forwarding the IP datagram directly from said first partition to said second partition of said multiple partitions without employing said network.

2. The method of claim 1, wherein said saving comprises saving each IP address of said at least one of said multiple partitions in an address resolution protocol (ARP) cache at said host-network interface as a HOME type IP address, and wherein said determining comprises employing said ARP cache to look up said destination IP address and determine whether said destination IP address comprises a HOME type IP address.

3. The method of claim 2, wherein said saving comprises saving in said ARP cache an IP address for each partition of said multiple partitions of the mainframe class data processing system, wherein each IP address of said mainframe class data processing system is saved in said ARP cache as a HOME type IP address.

4. The method of claim 2, further comprising saving in the ARP cache an IP address for at least one client coupled to said network, wherein said IP address of said at least one client is saved in said ARP cache as a NETWORK type IP address.

5. The method of claim 1, wherein said generating comprises generating said IP datagram and forwarding said IP datagram from said first partition without specifying a network environment for said IP datagram.

6. A system for network communications in a mainframe class data processing system having multiple partitions and a port to a network, said system comprising:

a host-network interface coupled between said multiple partitions and said network, said host-network interface being adapted to:

save at said host-network interface an internet protocol (IP) address of at least one of said multiple partitions of the mainframe class data processing system;

generate an IP datagram at a first partition of the multiple partitions to be forwarded to a second partition of the multiple partitions using a destination IP address; and

determine whether said destination IP address for said IP datagram comprises an IP address saved at the host-network interface for said at least one partition, and if so, forward the IP datagram directly from the first partition to the second partition of the multiple partitions without employing the network.

7. The system of claim 6, wherein said host-network interface being adapted to save comprises being adapted to save each IP address of said at least one of said multiple partitions in an address resolution protocol (ARP) cache at said host-network interface as a HOME type IP address, and wherein said host-network interface being adapted to determine comprises being adapted to employ said ARP cache to look up said destination IP address and determine whether said destination IP address comprises a HOME type IP address.

8. The system of claim 7, wherein said host-network interface being adapted to save comprises being adapted to save in said ARP cache an IP address for each partition of said multiple partitions of the mainframe class data processing system, wherein each IP address of said mainframe class data processing system is saved in said ARP cache as a HOME type IP address.

9. The system of claim 7, wherein said host-network interface is further adapted to save in the ARP cache an IP address for at least one client coupled to said network, wherein said IP address of said at least one client is saved in said ARP cache as a NETWORK type IP address.

10. The system of claim 6, wherein said host-network interface being adapted to generate comprises being adapted to generate said IP datagram and forward said IP datagram from said first partition without specifying a network environment for said IP datagram.

11. An article of manufacture comprising:

at least one computer usable medium having computer readable program code means embodied therein for causing network communications in a mainframe class data processing system having multiple partitions and a port to a network, the computer readable program code means in the article of manufacture comprising:

(i) computer readable program code means for causing a computer to effect saving at a host-network interface an internet protocol (IP) address of at least one of the multiple partitions of the mainframe class data processing system;

(ii) computer readable program code means for causing a computer to effect generating an IP datagram at a first partition of said multiple partitions to be forwarded to a second partition of said multiple partitions using a destination IP address; and

(iii) computer readable program code means for causing a computer to effect determining whether said destination IP address for said IP datagram comprises an IP address saved at said host-network interface for said at least one partition, and if so, forwarding the IP datagram directly from said first partition to said second partition of said multiple partitions without employing said network.

12. The article of manufacture of claim 11, wherein said computer readable program code means for causing a computer to effect saving comprises computer readable program code means for causing a computer to effect saving each IP address of said at least one of said multiple partitions in an address resolution protocol (ARP) cache at said host-network interface as a HOME type IP address, and wherein said computer readable program code means for causing a computer to effect determining comprises computer readable program code means for causing a computer to effect employing said ARP cache to look up said destination IP address and determine whether said destination IP address comprises a HOME type IP address.

13. The article of manufacture of claim 12, wherein said computer readable program code means for causing a computer to effect saving comprises computer readable program code means for causing a computer to effect saving in said ARP cache an IP address for each partition of said multiple partitions of the mainframe class data processing system, wherein each IP address of said mainframe class data processing system is saved in said ARP cache as a HOME type IP address.

14. The article of manufacture of claim 12, further comprising computer readable program code means for causing a computer to effect saving in the ARP cache an IP address for at least one client coupled to said network, wherein said IP address of said at least one client is saved in said ARP cache as a NETWORK type IP address.

15. The article of manufacture of claim 11, wherein said computer readable program code means for causing a computer to effect generating comprises computer readable program code means for causing a computer to effect generating said IP datagram and forwarding said IP datagram from said first partition without specifying a network environment for said IP datagram.

* * * * *

EXHIBIT 21

US006971002B2

(12) **United States Patent**
Austen et al.

(10) Patent No.: **US 6,971,002 B2**
(45) Date of Patent: **Nov. 29, 2005**

(54) **METHOD, SYSTEM, AND PRODUCT FOR BOOTING A PARTITION USING ONE OF MULTIPLE, DIFFERENT FIRMWARE IMAGES WITHOUT REBOOTING OTHER PARTITIONS**

(75) Inventors: **Christopher Harry Austen**, Austin, TX (US); **Van Hoa Lee**, Cedar Park, TX (US); **David R. Willoughby**, Austin, TX (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 541 days.

(21) Appl. No.: **09/925,584**

(22) Filed: **Aug. 9, 2001**

(65) **Prior Publication Data**

US 2003/0033512 A1 Feb. 13, 2003

(51) Int. Cl.[7] G06F 15/177; G06F 9/445; G06F 12/00

(52) U.S. Cl. 713/1; 713/2; 711/173

(58) Field of Search 713/1, 2; 711/162, 711/173

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,564,903 A 1/1986 Guyette et al. 364/300
4,843,541 A 6/1989 Bean et al. 364/200
5,345,590 A 9/1994 Ault et al. 395/650
6,430,663 B1 * 8/2002 Ding 711/162
6,684,343 B1 * 1/2004 Bouchier et al. 714/4
6,690,400 B1 * 2/2004 Moayyad et al. 345/779
6,725,317 B1 * 4/2004 Bouchier et al. 710/312

OTHER PUBLICATIONS

John Whitworth, Patition Magic and LILO, Oct. 5, 1998, Newsgroups: linux.redhat.install, pp. 2.*

IBM Technical Disclosure Bulletin, vol. 39, No. 12, Dec. 1996, "Hypervisor High Performance Synchronous Dispatch for Coupled Systems", one page.

IBM Technical Disclosure Bulletin, vol. 38, No. 04, Apr. 1995, "VM MPG Operating as a DRF Hypervisor as a First Level Guest Under PR/SM", p. 325.

IBM Technical Disclosure Bulletin, vol. 36, No. 03, Mar. 1993, "Sharing Read-Only Memory among Multiple Logical Partitions", pp. 303-304.

IBM Technical Disclosure Bulletin, vol. 39, No. 12, Dec. 1996, "Highly Parallel Coupling Facility Emulator/Router with Shadowed Link Buffers", 2 pages.

IBM Technical Disclosure Bulletin, vol. 39, No. 06, Jun. 1996, "Coordinating Multiple Server Partitions to Enter Power-Save State", pp. 235-239.

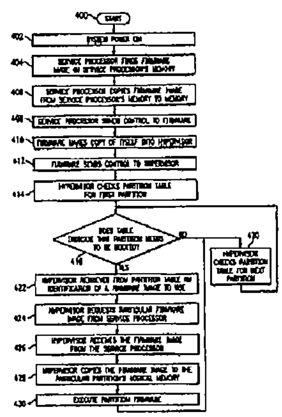
* cited by examiner

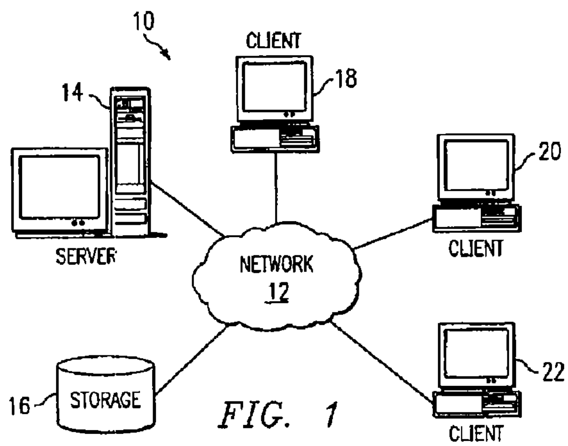
Primary Examiner—Thomas Lee
Assistant Examiner—Suresh K Suryawanshi
(74) *Attorney, Agent, or Firm*—Duke W. Yee; Mark E. McBurney; Lisa L. B. Yociss

(57) **ABSTRACT**

A method, system, and product within a logically partitioned computer system including multiple, different partitions are disclosed for booting a partition using one of multiple, different firmware images. These multiple, different firmware images are stored in the computer system. One of the partitions is rebooted utilizing one of the firmware images without rebooting other ones of the partitions.

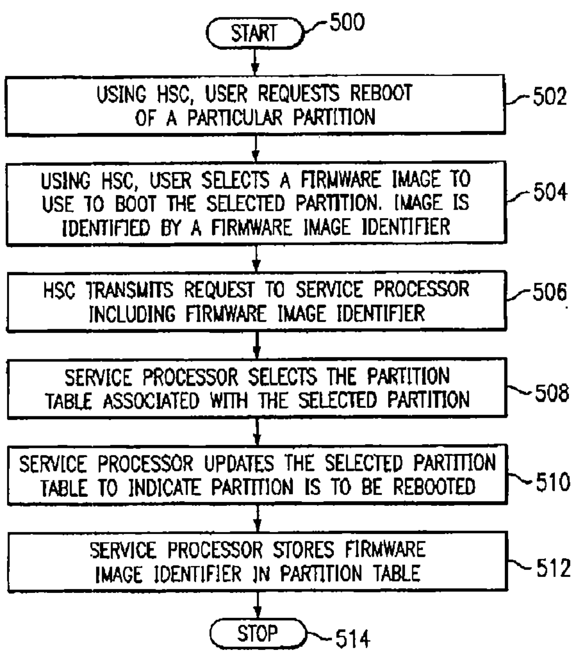
24 Claims, 4 Drawing Sheets





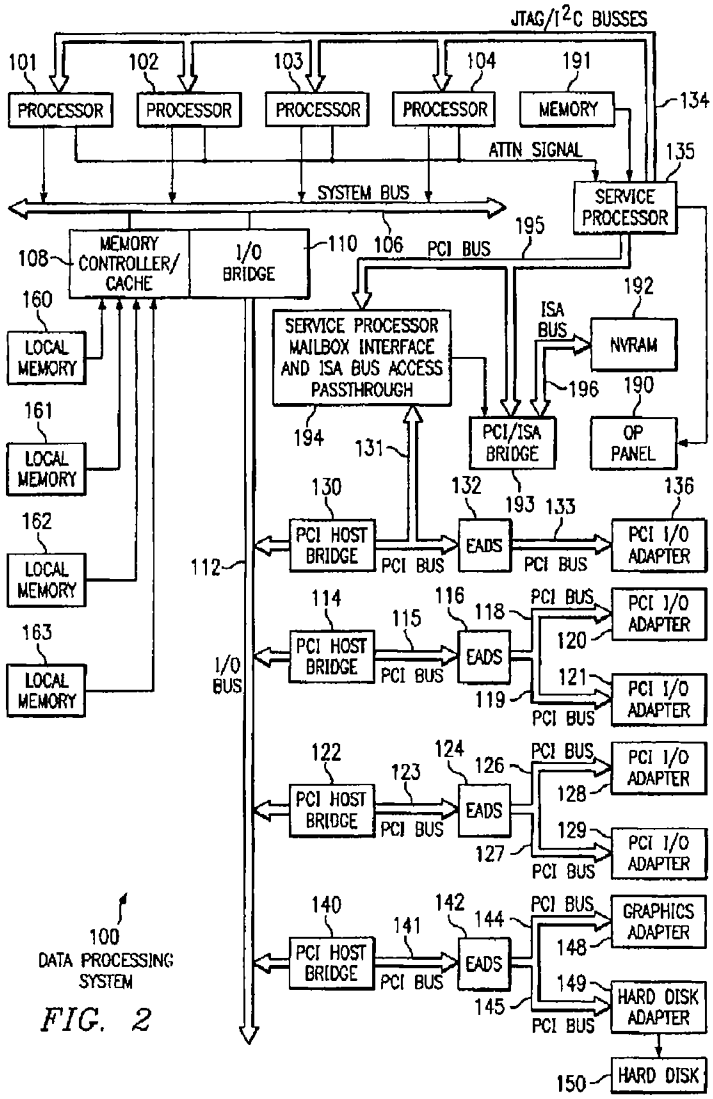
10
CLIENT
18
14
20
SERVER
NETWORK 12
CLIENT
22
16
STORAGE
CLIENT

FIG. 1



START
500
USING HSC, USER REQUESTS REBOOT OF A PARTICULAR PARTITION
502
USING HSC, USER SELECTS A FIRMWARE IMAGE TO USE TO BOOT THE SELECTED PARTITION. IMAGE IS IDENTIFIED BY A FIRMWARE IMAGE IDENTIFIER
504
HSC TRANSMITS REQUEST TO SERVICE PROCESSOR INCLUDING FIRMWARE IMAGE IDENTIFIER
506
SERVICE PROCESSOR SELECTS THE PARTITION TABLE ASSOCIATED WITH THE SELECTED PARTITION
508
SERVICE PROCESSOR UPDATES THE SELECTED PARTITION TABLE TO INDICATE PARTITION IS TO BE REBOOTED
510
SERVICE PROCESSOR STORES FIRMWARE IMAGE IDENTIFIER IN PARTITION TABLE
512
STOP
514

FIG. 5



JTAG/I2C BUSSES
101
102
103
104
191
PROCESSOR
PROCESSOR
PROCESSOR
PROCESSOR
MEMORY
134
ATTN SIGNAL
135
SYSTEM BUS
SERVICE PROCESSOR
108
MEMORY CONTROLLER/ CACHE
I/O BRIDGE
110
106
PCI BUS
195
160
LOCAL MEMORY
192
SERVICE PROCESSOR MAILBOX INTERFACE AND ISA BUS ACCESS PASSTHROUGH
ISA BUS
NVRAM
161
LOCAL MEMORY
194
131
PCI/ISA BRIDGE
196
190
OP PANEL
130
132
193
133
136
162
LOCAL MEMORY
PCI HOST BRIDGE
PCI BUS
EADS
PCI BUS
PCI I/O ADAPTER
112
114
115
116
118
PCI BUS
PCI I/O ADAPTER
163
LOCAL MEMORY
PCI HOST BRIDGE
PCI BUS
EADS
120
121
119
PCI BUS
PCI I/O ADAPTER
I/O BUS
122
123
124
126
PCI BUS
PCI I/O ADAPTER
PCI HOST BRIDGE
PCI BUS
EADS
128
129
127
PCI BUS
PCI I/O ADAPTER
100
DATA PROCESSING SYSTEM
140
141
142
144
PCI BUS
GRAPHICS ADAPTER
PCI HOST BRIDGE
PCI BUS
EADS
148
149
145
PCI BUS
HARD DISK ADAPTER
150
HARD DISK

FIG. 2

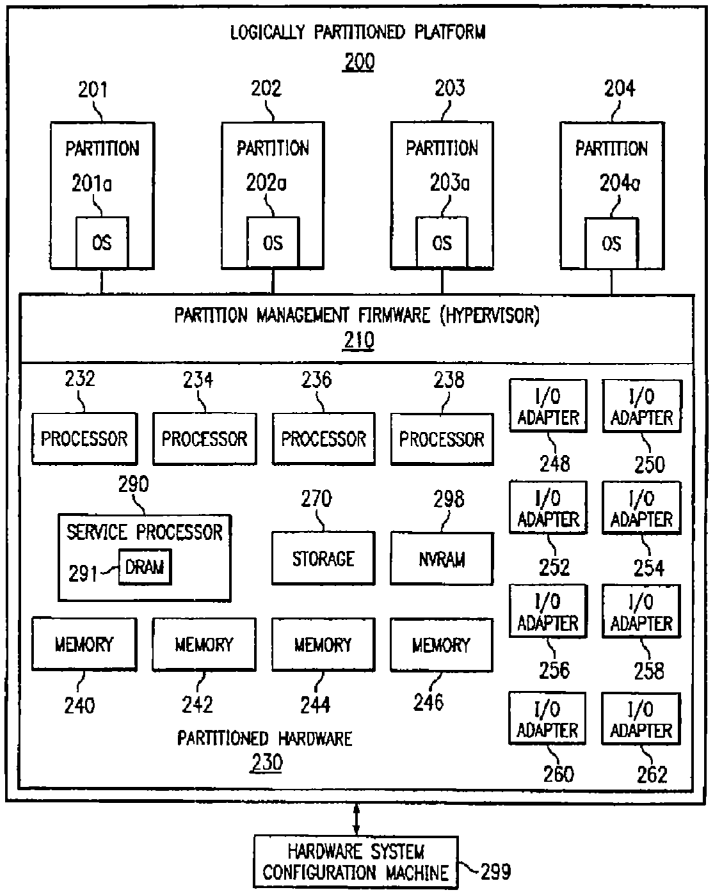

LOGICALLY PARTITIONED PLATFORM
200
201
202
203
204
PARTITION
201a
PARTITION
202a
PARTITION
203a
PARTITION
204a
OS
OS
OS
OS
PARTITION MANAGEMENT FIRMWARE (HYPERVISOR)
210
232
234
236
238
PROCESSOR
PROCESSOR
PROCESSOR
PROCESSOR
I/O ADAPTER
248
I/O ADAPTER
250
290
SERVICE PROCESSOR
291
DRAM
270
STORAGE
298
NVRAM
I/O ADAPTER
252
I/O ADAPTER
254
I/O ADAPTER
256
I/O ADAPTER
258
MEMORY
MEMORY
MEMORY
MEMORY
240
242
244
246
PARTITIONED HARDWARE
230
I/O ADAPTER
260
I/O ADAPTER
262
HARDWARE SYSTEM CONFIGURATION MACHINE
299

FIG. 3

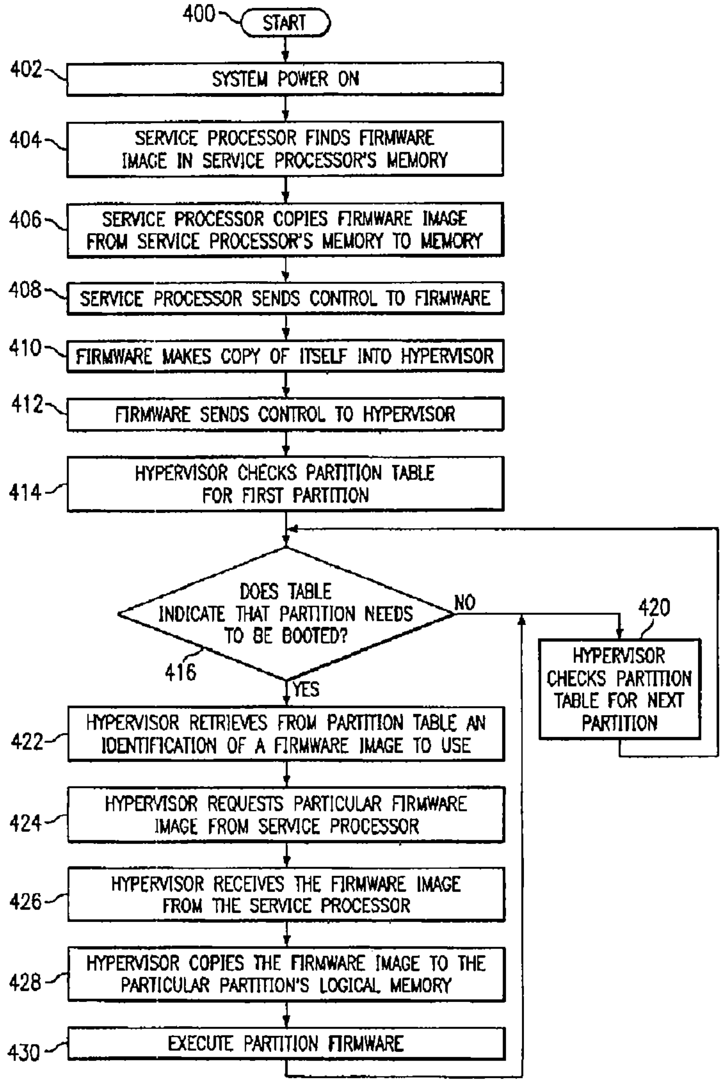

400
START
402
SYSTEM POWER ON
404
SERVICE PROCESSOR FINDS FIRMWARE IMAGE IN SERVICE PROCESSOR'S MEMORY
406
SERVICE PROCESSOR COPIES FIRMWARE IMAGE FROM SERVICE PROCESSOR'S MEMORY TO MEMORY
408
SERVICE PROCESSOR SENDS CONTROL TO FIRMWARE
410
FIRMWARE MAKES COPY OF ITSELF INTO HYPERVISOR
412
FIRMWARE SENDS CONTROL TO HYPERVISOR
414
HYPERVISOR CHECKS PARTITION TABLE FOR FIRST PARTITION
DOES TABLE INDICATE THAT PARTITION NEEDS TO BE BOOTED?
416
NO
420
HYPERVISOR CHECKS PARTITION TABLE FOR NEXT PARTITION
YES
422
HYPERVISOR RETRIEVES FROM PARTITION TABLE AN IDENTIFICATION OF A FIRMWARE IMAGE TO USE
424
HYPERVISOR REQUESTS PARTICULAR FIRMWARE IMAGE FROM SERVICE PROCESSOR
426
HYPERVISOR RECEIVES THE FIRMWARE IMAGE FROM THE SERVICE PROCESSOR
428
HYPERVISOR COPIES THE FIRMWARE IMAGE TO THE PARTICULAR PARTITION'S LOGICAL MEMORY
430
EXECUTE PARTITION FIRMWARE

FIG. 4

METHOD, SYSTEM, AND PRODUCT FOR BOOTING A PARTITION USING ONE OF MULTIPLE, DIFFERENT FIRMWARE IMAGES WITHOUT REBOOTING OTHER PARTITIONS

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates generally to the field of computer systems and, more specifically to computer systems having multiple, logical partitions. Still more particularly, the present invention relates to a logically partitioned computer system, method, and product for booting one of the partitions using one of multiple, different firmware images.

2. Description of Related Art

A logical partitioning option (LPAR) within a data processing system (platform) allows multiple copies of a single operating system (OS) or multiple heterogeneous operating systems to be simultaneously run on a single data processing system hardware platform. A partition, within which an operating system image runs, is assigned a non-overlapping subset of the platform's hardware resources. These platform allocable resources include one or more architecturally distinct processors with their interrupt management area, regions of system memory, and input/output (I/O) adapter bus slots. The partition's resources are represented by its own open firmware device tree to the OS image.

Each distinct OS or image of an OS running within the platform is protected from each other such that software errors on one logical partition can not affect the correct operation of any of the other partitions. This is provided by allocating a disjoint set of platform resources to be directly managed by each OS image and by providing mechanisms for ensuring that the various images can not control any resources that have not been allocated to it. Furthermore, software errors in the control of an operating system's allocated resources are prevented from affecting the resources of any other image. Thus, each image of the OS (or each different OS) directly controls a distinct set of allocable resources within the platform.

Many logically partitioned systems make use of a hypervisor. A hypervisor is a layer of privileged software between the hardware and logical partitions that manages and enforces partition protection boundaries. The hypervisor is also referred to herein as partition management firmware or firmware. The hypervisor is responsible for configuring, servicing, and running multiple logical systems on the same physical hardware. The hypervisor is typically responsible for allocating resources to a partition, installing an operating system in a partition, starting and stopping the operating system in a partition, dumping main storage of a partition, communicating between partitions, and providing other functions. In order to implement these functions, the hypervisor also has to implement its own low level operations like main storage management, synchronization primitives, I/O facilities, heap management, and other functions.

Currently, only a single firmware image can exist within a logically partitioned computer system. This firmware image is used to boot each partition. Each partition, thus, boots from the same image. When the firmware image is modified, the entire boot process must be repeated with each partition being rebooted from the same modified firmware image.

Therefore, a need exists for a logically partitioned system, method, and product for maintaining multiple, different firmware images, and booting only one of the partitions using one of these firmware images, wherein there is no need to reboot the entire system.

SUMMARY OF THE INVENTION

A method, system, and product within a logically partitioned computer system including multiple, different partitions are disclosed for booting a partition using one of multiple, different firmware images. These multiple, different firmware images are stored in the computer system. One of the partitions is rebooted utilizing one of the firmware images without rebooting other ones of the partitions.

The above as well as additional objectives, features, and advantages of the present invention will become apparent in the following detailed written description.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself, however, as well as a preferred mode of use, further objectives and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

FIG. 1 is a pictorial representation which depicts a data processing system in which the present invention may be implemented in accordance with a preferred embodiment of the present invention;

FIG. 2 is a more detailed block diagram of a data processing system in which the present invention may be implemented in accordance with the present invention;

FIG. 3 is a block diagram of an exemplary logically partitioned platform in which the present invention may be implemented;

FIG. 4 illustrates a high level flow chart which depicts booting one partition using one of a plurality of different firmware images maintained by a logically partitioned computer system without the need for rebooting the entire system in accordance with the present invention; and

FIG. 5 depicts a high level flow chart which illustrates selecting one of a plurality of different firmware images to use to boot one of multiple, different partitions in a logically partitioned computer system in accordance with the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

A preferred embodiment of the present invention and its advantages are better understood by referring to the figures, like numerals being used for like and corresponding parts of the accompanying figures.

The present invention is a method, system, and product for maintaining a plurality of different firmware images, and for rebooting one of a plurality of different partitions using one of these images without rebooting other ones of the partitions. These multiple, different firmware images are stored in non-volatile memory in a logically partitioned computer system.

Each partition is associated with its own, unique partition table. In addition to other information stored in a partition table, an indicator within the partition table is used to indicate whether the partition associated with the table needs

to be rebooted. Also stored within a partition table is an identifier which identifies one of the different firmware images.

When the indicator indicates that the partition associated with the partition table having the indicator needs to be rebooted, the computer system will copy the firmware image identified by the identifier stored in that partition table to the associated partition. This firmware image is then executed in order to boot only that partition. Other partitions are unaffected by the copying of the firmware image and the rebooting of the partition.

FIG. 1 depicts a pictorial representation of a network of data processing systems in which the present invention may be implemented. Network data processing system 10 is a network of computers in which the present invention may be implemented. Network data processing system 10 contains a network 12, which is the medium used to provide communications links between various devices and computers connected together within network data processing system 10. Network 12 may include connections, such as wire, wireless communication links, or fiber optic cables.

In the depicted example, a server 14 is connected to network 12 along with storage unit 16. In addition, clients 18, 20, and 22 also are connected to network 12. Network 12 may include permanent connections, such as wire or fiber optic cables, or temporary connections made through telephone connections. The communications network 12 also can include other public and/or private wide area networks, local area networks, wireless networks, data communication networks or connections, intranets, routers, satellite links, microwave links, cellular or telephone networks, radio links, fiber optic transmission lines, ISDN lines, T1 lines, DSL, etc. In some embodiments, a user device may be connected directly to a server 14 without departing from the scope of the present invention. Moreover, as used herein, communications include those enabled by wired or wireless technology.

Clients 18, 20, and 22 may be, for example, personal computers, portable computers, mobile or fixed user stations, workstations, network terminals or servers, cellular telephones, kiosks, dumb terminals, personal digital assistants, two-way pagers, smart phones, information appliances, or network computers. For purposes of this application, a network computer is any computer, coupled to a network, which receives a program or other application from another computer coupled to the network.

In the depicted example, server 14 provides data, such as boot files, operating system images, and applications to clients 18–22. Clients 18, 20, and 22 are clients to server 14. Network data processing system 10 may include additional servers, clients, and other devices not shown. In the depicted example, network data processing system 10 is the Internet with network 12 representing a worldwide collection of networks and gateways that use the TCP/IP suite of protocols to communicate with one another. At the heart of the Internet is a backbone of high-speed data communication lines between major nodes or host computers, consisting of thousands of commercial, government, educational and other computer systems that route data and messages. Of course, network data processing system 10 also may be implemented as a number of different types of networks, such as for example, an intranet, a local area network (LAN), or a wide area network (WAN). FIG. 1 is intended as an example, and not as an architectural limitation for the present invention.

FIG. 2 is a more detailed block diagram of a data processing system in which the present invention may be

implemented. Data processing system 100 may be a symmetric multiprocessor (SMP) system including a plurality of processors 101, 102, 103, and 104 connected to system bus 106. For example, data processing system 100 may be an IBM RS/6000, a product of International Business Machines Corporation in Armonk, N.Y., implemented as a server within a network. Alternatively, a single processor system may be employed. Also connected to system bus 106 is memory controller/cache 108, which provides an interface to a plurality of local memories 160–163. I/O bus bridge 110 is connected to system bus 106 and provides an interface to I/O bus 112. Memory controller/cache 108 and I/O bus bridge 110 may be integrated as depicted.

Data processing system 100 is a logically partitioned data processing system. Thus, data processing system 100 may have multiple heterogeneous operating systems (or multiple instances of a single operating system) running simultaneously. Each of these multiple operating systems may have any number of software programs executing within it. Data processing system 100 is logically partitioned such that different I/O adapters 120–121, 128–129, 136, and 148–149 may be assigned to different logical partitions.

Thus, for example, suppose data processing system 100 is divided into three logical partitions, P1, P2, and P3. Each of I/O adapters 120–121, 128–129, 136, and 148–149, each of processors 101–104, and each of local memories 160–163 is assigned to one of the three partitions. For example, processor 101, memory 160, and I/O adapters 120, 128, and 129 may be assigned to logical partition P1; processors 102–103, memory 161, and I/O adapters 121 and 136 may be assigned to partition P2; and processor 104, memories 162–163, and I/O adapters 148–149 may be assigned to logical partition P3.

Each operating system executing within data processing system 100 is assigned to a different logical partition. Thus, each operating system executing within data processing system 100 may access only those I/O units that are within its logical partition.

Peripheral component interconnect (PCI) Host bridge 114 connected to I/O bus 112 provides an interface to PCI local bus 115. A number of Input/Output adapters 120–121 may be connected to PCI bus 115. Typical PCI bus implementations will support between four and eight I/O adapters (i.e. expansion slots for add-in connectors). Each I/O Adapter 120–121 provides an interface between data processing system 100 and input/output devices such as, for example, other network computers, which are clients to data processing system 100.

An additional PCI host bridge 122 provides an interface for an additional PCI bus 123. PCI bus 123 is connected to a plurality of PCI I/O adapters 128–129 by a PCI bus 126–127. Thus, additional I/O devices, such as, for example, modems or network adapters may be supported through each of PCI I/O adapters 128–129. In this manner, data processing system 100 allows connections to multiple network computers.

A memory mapped graphics adapter 148 may be connected to I/O bus 112 through PCI Host Bridge 140 and EADS 142 (PCI-PCI bridge) via PCI buses 144 and 145 as depicted. Also, a hard disk 150 may also be connected to I/O bus 112 through PCI Host Bridge 140 and EADS 142 via PCI buses 141 and 145 as depicted.

A PCI host bridge 130 provides an interface for a PCI bus 131 to connect to I/O bus 112. PCI bus 131 connects PCI host bridge 130 to the service processor mailbox interface and ISA bus access pass-through logic 194 and EADS 132. The ISA bus access pass-through logic 194 forwards PCI

accesses destined to the PCI/ISA bridge 193. The NV-RAM storage is connected to the ISA bus 196. The Service processor 135 is coupled to the service processor mailbox interface 194 through its local PCI bus 195. Service processor 135 is also connected to processors 101–104 via a plurality of JTAG/I^{2}C buses 134. JTAG/I^{2}C buses 134 are a combination of JTAG/scan busses (see IEEE 1149.1) and Phillips I^{2}C busses. However, alternatively, JTAG/I^{2}C buses 134 may be replaced by only Phillips I^{2}C busses or only JTAG/scan busses. All SP-ATTN signals of the host processors 101, 102, 103, and 104 are connected together to an interrupt input signal of the service processor. The service processor 135 has its own local memory 191, and has access to the hardware op-panel 190.

When data processing system 100 is initially powered up, service processor 135 uses the JTAG/scan buses 134 to interrogate the system (Host) processors 101–104, memory controller 108, and I/O bridge 110. At completion of this step, service processor 135 has an inventory and topology understanding of data processing system 100. Service processor 135 also executes Built-In-Self-Tests (BISTs), Basic Assurance Tests (BATs), and memory tests on all elements found by interrogating the system processors 101–104, memory controller 108, and I/O bridge 110. Any error information for failures detected during the BISTs, BATs, and memory tests are gathered and reported by service processor 135.

If a meaningful/valid configuration of system resources is still possible after taking out the elements found to be faulty during the BISTs, BATs, and memory tests, then data processing system 100 is allowed to proceed to load executable code into local (Host) memories 160–163. Service processor 135 then releases the Host processors 101–104 for execution of the code loaded into Host memory 160–163. While the Host processors 101–104 are executing code from respective operating systems within the data processing system 100, service processor 135 enters a mode of monitoring and reporting errors. The type of items monitored by service processor include, for example, the cooling fan speed and operation, thermal sensors, power supply regulators, and recoverable and non-recoverable errors reported by processors 101–104, memories 160–163, and bus-bridge controller 110.

Service processor 135 is responsible for saving and reporting error information related to all the monitored items in data processing system 100. Service processor 135 also takes action based on the type of errors and defined thresholds. For example, service processor 135 may take note of excessive recoverable errors on a processor's cache memory and decide that this is predictive of a hard failure. Based on this determination, service processor 135 may mark that resource for reconfiguration during the current running session and future Initial Program Loads (IPLs). IPLs are also sometimes referred to as a "boot" or "bootstrap".

Those of ordinary skill in the art will appreciate that the hardware depicted in FIG. 2 may vary. For example, other peripheral devices, such as optical disk drives and the like, also may be used in addition to or in place of the hardware depicted. The depicted example is not meant to imply architectural limitations with respect to the present invention.

FIG. 3 is a block diagram of an exemplary logically partitioned platform in which the present invention may be implemented. Logically partitioned platform 200 includes partitioned hardware 230, partition management firmware, also called a hypervisor 210, and partitions 201–204. Operating systems 201a–204a exist within partitions 201–204. Operating systems 201a–204a may be multiple copies of a single operating system or multiple heterogeneous operating systems simultaneously run on platform 200.

Partitioned hardware 230 includes a plurality of processors 232–238, a plurality of system memory units 240–246, a plurality of input/output (I/O) adapters 248–262, and a storage unit 270. Each of the processors 242–248, memory units 240–246, NV-RAM storage 298f and I/O adapters 248–262 may be assigned to one of multiple partitions 201–204.

Partitioned hardware 230 also includes service processor 290. A non-volatile memory device 291, such as a DRAM device, is included within service processor 291. The partition tables and firmware images described herein, as well as other information, are stored within service processor memory 291.

Partition management firmware (hypervisor) 210 performs a number of functions and services for partitions 201–204 to create and enforce the partitioning of logically partitioned platform 200. Hypervisor 210 is a firmware implemented virtual machine identical to the underlying hardware. Firmware is "software" stored in a memory chip that holds its content without electrical power, such as, for example, read-only memory (ROM), programmable ROM (PROM), erasable programmable ROM (EPROM), electrically erasable programmable ROM (EEPROM), and nonvolatile random access memory (non-volatile RAM). Thus, hypervisor 210 allows the simultaneous execution of independent OS images 201a–204a by virtualizing all the hardware resources of logically partitioned platform 200. Hypervisor 210 may attach I/O devices through I/O adapters 248–262 to single virtual machines in an exclusive mode for use by one of OS images 201a–204a.

A hardware system configuration (HSC) machine 299 may be coupled to data processing system 100 which includes logically partitioned platform 200. HSC 299 is a separate computer system that is coupled to service processor 290 and may be used by a user to control various functions of data processing system 100 through service processor 290. HSC 299 includes a graphical user interface (GUI) which may be used by a user to select a partition to be rebooted. Further, a listing of different firmware images that are stored within service processor memory 291 may be presented to the user utilizing the graphical user interface of HSC 299. The user may then select one of the listed firmware images to use to boot the selected partition as described below.

When a user selects a partition, HSC 299 transmits a request to service processor 290 to have service processor 290 update the partition table associated with the selected partition. Service processor 290 updates the partition table by setting an indicator within the table to indicate that the associated partition needs to be rebooted. In addition, HSC 299 transmits an identifier to service processor 290 which identifies the particular firmware image selected by the user. Service processor 290 then stores this identifier within the partition table associated with the selected partition.

As described in more detail below, hypervisor 210 routinely checks each partition table to determine a current state of the indicator stored in each table. When hypervisor 210 finds an indicator that indicates a partition needs to be rebooted, hypervisor 210 copies the firmware image identified within that partition table to the logical memory of the partition associated with the partition table. That firmware image is then executed within the partition causing only that partition to be rebooted. Other partitions are unaffected by this process.

FIG. 4 illustrates a high level flow chart which depicts booting one partition using one of a plurality of different firmware images maintained by a logically partitioned computer system without the need for rebooting the entire system in accordance with the present invention. The process starts as depicted by block 400 and thereafter passes to block 402 which illustrates the system being powered-on. Next, block 404 depicts service processor 290 finding a firmware image in the service processor's memory 291 to use when first powering-up the entire computer system.

Block 406 then illustrates service processor 290 copying the firmware image from the service processor's memory 291 to a memory device, such as memory 240–246. The process then passes to block 408 which depicts service processor 290 sending control to the firmware copied into memory 240–246. Next, block 410 illustrates the firmware in memory 240–246 copying itself into hypervisor 210.

Thereafter, block 412 depicts firmware in memory 240–246 sending control to hypervisor 210. Next, block 414 illustrates hypervisor 210 checking a partition table for a first partition. The process then passes to block 416 which depicts a determination of whether or not the partition table indicates that the partition associated with the partition table needs to be rebooted. If a determination is made that the partition associated with the partition table does not need to be rebooted, the process passes to block 420 which illustrates hypervisor 210 checking a partition table associated with a next partition. The process then passes back to block 416.

Referring again to block 416, if a determination is made that the partition associated with the partition table does need to be rebooted, the process passes to block 422 which illustrates hypervisor 210 retrieving from the partition table an identification of one of multiple, different firmware images to use. Next, block 424 depicts hypervisor 210 requesting the firmware image identified by the identification retrieved from the partition table.

The process then passes to block 426 which illustrates hypervisor 210 receiving the requested firmware image. Next, block 428 depicts hypervisor 210 copying the requested firmware image to the particular partition's logical memory. Thereafter, block 430 illustrates executing the firmware image copied into the partition. In this manner, only the particular partition is rebooted. The process then passes to block 420.

FIG. 5 depicts a high level flow chart which illustrates selecting one of a plurality of different firmware images to use to boot one of multiple, different partitions in a logically partitioned computer system in accordance with the present invention. The process starts as depicted by block 500 and thereafter passes to block 502 which illustrates permitting a user to request a reboot of a particular partition using the hardware system configuration (HSC) machine 299. Next, block 504 depicts permitting the user, who is using the HSC machine 299, to select one of multiple, different firmware images to use to boot the selected partition. The image is identified by a firmware image identifier, which is then included in the request.

The process then passes to block 506 which illustrates HSC 299 transmitting the request made by the user to service processor 290. The request identifies a particular partition and includes a firmware image identifier. Next, block 508 depicts service processor 290 selecting the partition table that is associated with the partition identified by the request. Thereafter, block 510 illustrates service processor 290 updating the selected partition table to indicate that the partition associated with the table is to be rebooted. Block 512, then, depicts service processor 290 storing the firmware image identifier in the selected partition table. The process then terminates as illustrated by block 514.

It is important to note that while the present invention has been described in the context of a fully functioning data processing system, those of ordinary skill in the art will appreciate that the processes of the present invention are capable of being distributed in the form of a computer readable medium of instructions and a variety of forms and that the present invention applies equally regardless of the particular type of signal bearing media actually used to carry out the distribution. Examples of computer readable media include recordable-type media, such as a floppy disk, a hard disk drive, a RAM, CD-ROMs, DVD-ROMs, and transmission-type media, such as digital and analog communications links, wired or wireless communications links using transmission forms, such as, for example, radio frequency and light wave transmissions. The computer readable media may take the form of coded formats that are decoded for actual use in a particular data processing system.

The description of the present invention has been presented for purposes of illustration and description, and is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art. The embodiment was chosen and described in order to best explain the principles of the invention, the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.

What is claimed is:

1. A method in a logically partitioned computer system including a plurality of different partitions, said method comprising the steps of:
 storing a plurality of different firmware images in said computer system, each one of said plurality of different firmware images capable of being executed during a power-on process to boot said computer system; and
 rebooting one of said plurality of partitions utilizing one of said plurality of firmware images without rebooting other ones of said plurality of partitions, rebooting said one of said plurality of partitions utilizing said one of said plurality of firmware images prior to booting an operating system in said one of said plurality of partitions.

2. The method according to claim 1, further comprising the step of selecting said one of said plurality of firmware images to use to reboot said one of said plurality of partitions.

3. The method according to claim 1, further comprising the step of associating a different, unique firmware image identifier with each of said plurality of firmware images.

4. The method according to claim 1, further comprising the steps of:
 associating a different, unique firmware image identifier with each of said plurality of firmware images;
 associating a different partition table with each one of said plurality of partitions;
 providing an indicator within each said different partition table, said indicator indicating whether one of said plurality of partitions that is associated with said partition table needs to be rebooted; and
 providing an identifier within each said different partition table, said identifier identifying one of said plurality of firmware images.

5. The method according to claim 1, further comprising the steps of:

routinely checking each said partition table to determine whether said indicator included within each said partition table indicates that one of said plurality of partitions associated with each said partition table is to be rebooted;

in response to a determination that an indicator indicates that one of said plurality of partitions needs to be rebooted, rebooting said one of said plurality of partitions having said indicator that indicates said need to be rebooted.

6. The method according to claim 5, further comprising the steps of:

retrieving an identifier from said partition table that includes said indicator that indicates said need to be rebooted, said identifier one of said plurality of firmware images; and

rebooting only said one of said plurality of partitions that includes said indicator that indicates said need to be rebooted utilizing said identifier retrieved from said partition table associated with said one of said plurality of partitions.

7. The method according to claim 1, further comprising the steps of:

providing a listing of said plurality of partitions;

providing a listing of said plurality of different firmware images;

receiving a selection of one of said plurality of partitions that is to be rebooted; and

receiving a selection of one of said plurality of firmware images to use to reboot said selected one of said plurality of partitions.

8. The method according to claim 7, further comprising the steps of:

setting an indicator in a partition table associated with said selected one of said plurality of partitions, said indicator indicating that said one of said plurality of partitions has been selected to be rebooted; and

storing an identification of said selected one of said plurality of firmware images in said partition table associated with said selected one of said plurality of partitions.

9. A computer program product stored in a computer recordable-type media in a logically partitioned computer system including a plurality of different partitions, comprising:

instruction means for storing a plurality of different firmware images in said computer system, each one of said plurality of different firmware images capable of being executed during a power-on process to boot said computer system; and

instruction means for rebooting one of said plurality of partitions utilizing one of said plurality of firmware images without rebooting other ones of said plurality of partitions, rebooting said one of said plurality of partitions utilizing said one of said plurality of firmware images prior to booting an operating system in said one of said plurality of partitions.

10. The product according to claim 9, further comprising instruction means for selecting said one of said plurality of firmware images to use to reboot said one of said plurality of partitions.

11. The product according to claim 9, further comprising instruction means for associating a different, unique firmware image identifier with each of said plurality of firmware images.

12. The product according to claim 9, further comprising:

instruction means for associating a different, unique firmware image identifier with each of said plurality of firmware images;

instruction means for associating a different partition table with each one of said plurality of partitions;

instruction means for providing an indicator within each said different partition table, said indicator indicating whether one of said plurality of partitions that is associated with said partition table needs to be rebooted; and

instruction means for providing an identifier within each said different partition table, said identifier identifying one of said plurality of firmware images.

13. The product according to claim 9, further comprising:

instruction means for routinely checking each said partition table to determine whether said indicator included within each said partition table indicates that one of said plurality of partitions associated with each said partition table is to be rebooted;

in response to a determination that an indicator indicates that one of said plurality of partitions needs to be rebooted, instruction means for rebooting said one of said plurality of partitions having said indicator that indicates said need to be rebooted.

14. The product according to claim 13, further comprising:

instruction means for retrieving an identifier from said partition table that includes said indicator that indicates said need to be rebooted, said identifier one of said plurality of firmware images; and

instruction means for rebooting only said one of said plurality of partitions that includes said indicator that indicates said need to be rebooted utilizing said identifier retrieved from said partition table associated with said one of said plurality of partitions.

15. The product according to claim 9, further comprising:

instruction means for providing a listing of said plurality of partitions;

instruction means for providing a listing of said plurality of different firmware images;

instruction means for receiving a selection of one of said plurality of partitions that is to be rebooted; and

instruction means for receiving a selection of one of said plurality of firmware images to use to reboot said selected one of said plurality of partitions.

16. The product according to claim 15, further comprising:

instruction means for setting an indicator in a partition table associated with said selected one of said plurality of partitions, said indicator indicating that said one of said plurality of partitions has been selected to be rebooted; and

instruction means for storing an identification of said selected one of said plurality of firmware images in said partition table associated with said selected one of said plurality of partitions.

17. A logically partitioned computer system including a plurality of different partitions, comprising:

a plurality of different firmware images being stored in said computer system, each one of said plurality of different firmware images capable of being executed during a power-on process to boot said computer system; and

said computer system for rebooting one of said plurality of partitions utilizing one of said plurality of firmware images without rebooting other ones of said plurality of

partitions, rebooting said one of said plurality of partitions utilizing said one of said plurality of firmware images prior to booting an operating system in said one of said plurality of partitions.

18. The system according to claim 17, further comprising said one of said plurality of firmware images being selected to use to reboot said one of said plurality of partitions.

19. The system according to claim 17, further comprising a different, unique firmware image identifier being associated with each of said plurality of firmware images.

20. The system according to claim 17, further comprising:

a different, unique firmware image identifier being associated with each of said plurality of firmware images;

a different partition table being associated with each one of said plurality of partitions;

an indicator being provided within each said different partition table, said indicator indicating whether one of said plurality of partitions that is associated with said partition table needs to be rebooted; and

an identifier being provided within each said different partition table, said identifier identifying one of said plurality of firmware images.

21. The system according to claim 17, further comprising:

said computer system for routinely checking each said partition table to determine whether said indicator included within each said partition table indicates that one of said plurality of partitions associated with each said partition table is to be rebooted;

in response to a determination that an indicator indicates that one of said plurality of partitions needs to be rebooted, said computer system for rebooting said one

of said plurality of partitions having said indicator that indicates said need to be rebooted.

22. The system according to claim 21, further comprising:

an identifier being retrieved from said partition table that includes said indicator that indicates said need to be rebooted, said identifier one of said plurality of firmware images; and

said computer system for rebooting only said one of said plurality of partitions that includes said indicator that indicates said need to be rebooted utilizing said identifier retrieved from said partition table associated with said one of said plurality of partitions.

23. The system according to claim 17, further comprising:

a listing of said plurality of partitions;

a listing of said plurality of different firmware images;

means for receiving a selection of one of said plurality of partitions that is to be rebooted; and

means for receiving a selection of one of said plurality of firmware images to use to reboot said selected one of said plurality of partitions.

24. The system according to claim 23, further comprising:

an indicator in a partition table associated with said selected one of said plurality of partitions being set, said indicator indicating that said one of said plurality of partitions has been selected to be rebooted; and

an identification of said selected one of said plurality of firmware images being stored in said partition table associated with said selected one of said plurality of partitions.

* * * * *

EXHIBIT 22

US005414851A

United States Patent [19]

Brice, Jr. et al.

[11] Patent Number: 5,414,851

[45] Date of Patent: May 9, 1995

[54] METHOD AND MEANS FOR SHARING I/O RESOURCES BY A PLURALITY OF OPERATING SYSTEMS

[75] Inventors: Frank W. Brice, Jr., Hurley; Joseph C. Elliott, Hopewell Junction; Kenneth J. Fredericks; Robert E. Galbraith, both of Poughkeepsie; Marten J. Halma, Poughquag; Roger E. Hough, Highland; Suzanne M. John, Poughkeepsie; Paul A. Malinowski, Poughkeepsie; Allan S. Meritt, Poughkeepsie; Kenneth J. Oakes, Wappingers Falls; John C. Rathjen, Jr., Rhinebeck, all of N.Y.; Martin W. Sachs, Westport, Conn.; David E. Stucki; Leslie W. Wyman, both of Poughkeepsie, N.Y.

[73] Assignee: International Business Machines Corporation, Armonk, N.Y.

[21] Appl. No.: 898,867

[22] Filed: Jun. 15, 1992

[51] Int. Cl.⁶ G06F 3/00

[52] U.S. Cl. 395/650; 395/275; 364/DIG. 1; 364/280; 364/281.6; 364/281.3

[58] Field of Search 395/650, 700, 275

[56] References Cited

U.S. PATENT DOCUMENTS

4,084,224	4/1978	Appell et al.	364/200
4,253,145	2/1981	Goldberg	395/500
4,564,903	1/1986	Guyette et al.	364/200
4,843,541	6/1989	Bean et al.	364/200
4,885,681	12/1989	Umeno et al.	395/375
4,967,342	10/1990	Lent et al.	364/200
5,170,472	12/1992	Cwiakala et al.	395/275
5,251,317	10/1993	Iizuka et al.	395/650
5,297,262	3/1994	Cox et al.	395/325

FOREIGN PATENT DOCUMENTS

0301275	1/1989	European Pat. Off.	G06F 9/46

Primary Examiner—Kevin A. Kriess
Attorney, Agent, or Firm—Bernard M. Goldman

[57] ABSTRACT

Provides a method for increasing the connectivity of I/O resources to a multiplicity of operating systems (OSs) running in different resource partitions of a computer electronic complex (CEC) to obtain sharing of the I/O resources among the OSs of the CEC, including channels, subchannels (devices), and control units (CUs). The invention provides image identifiers (IIDs) for assigning resources to the different OSs. Each shared I/O resource has a sharing set of control blocks (CBs) in which a respective CB is assigned to (and located by) a respective IID of one of the OSs. Each of the CBs in a sharing set provides a different image of the same I/O resource. The different CB images are independently set to different states by I/O operations for the different OSs, so that the OSs can independently share the same I/O resource.

30 Claims, 16 Drawing Sheets

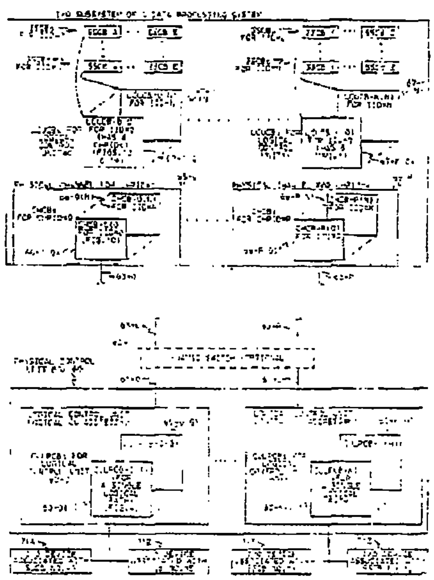
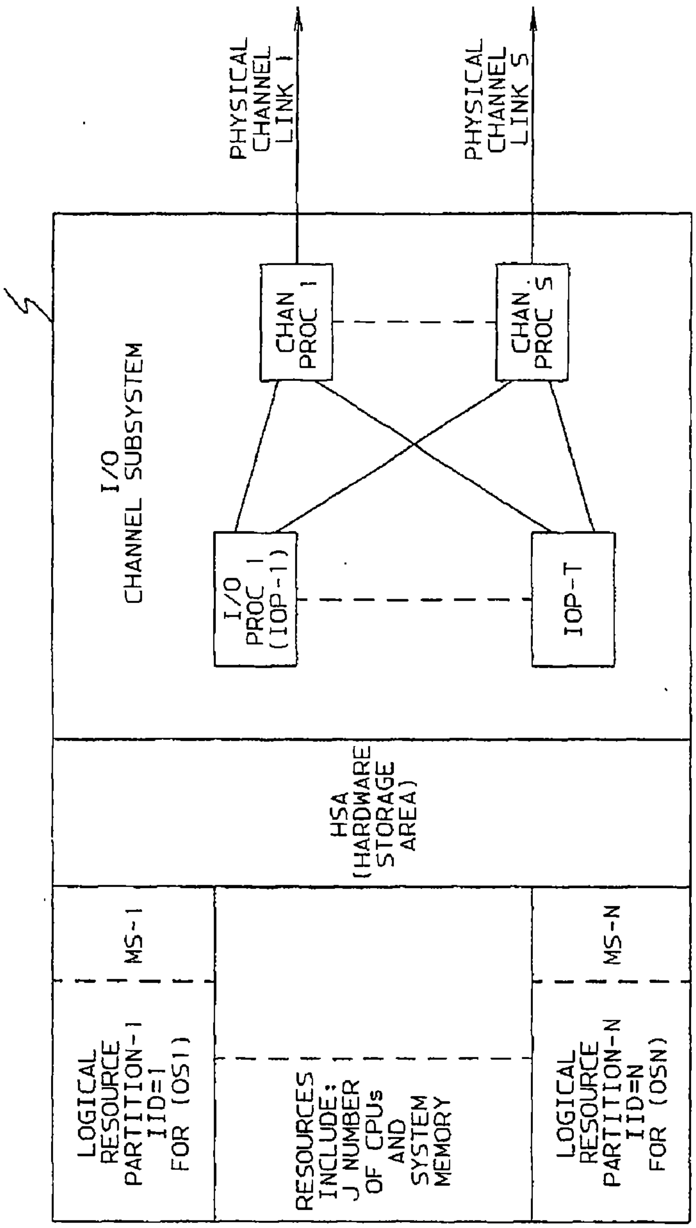


FIG. 1


CEC
I/O
CHANNEL SUBSYSTEM
CHAN PROC 1
CHAN. PROC S
PHYSICAL CHANNEL LINK 1
PHYSICAL CHANNEL LINK S
I/O PROC 1 (IOP-1)
IOP-T
HSA (HARDWARE STORAGE AREA)
MS-1
LOGICAL RESOURCE PARTITION-1 IID=1 FOR (OS1)
RESOURCES INCLUDE: J NUMBER OF CPUs AND SYSTEM MEMORY
MS-N
LOGICAL RESOURCE PARTITION-N IID=N FOR (OSN)

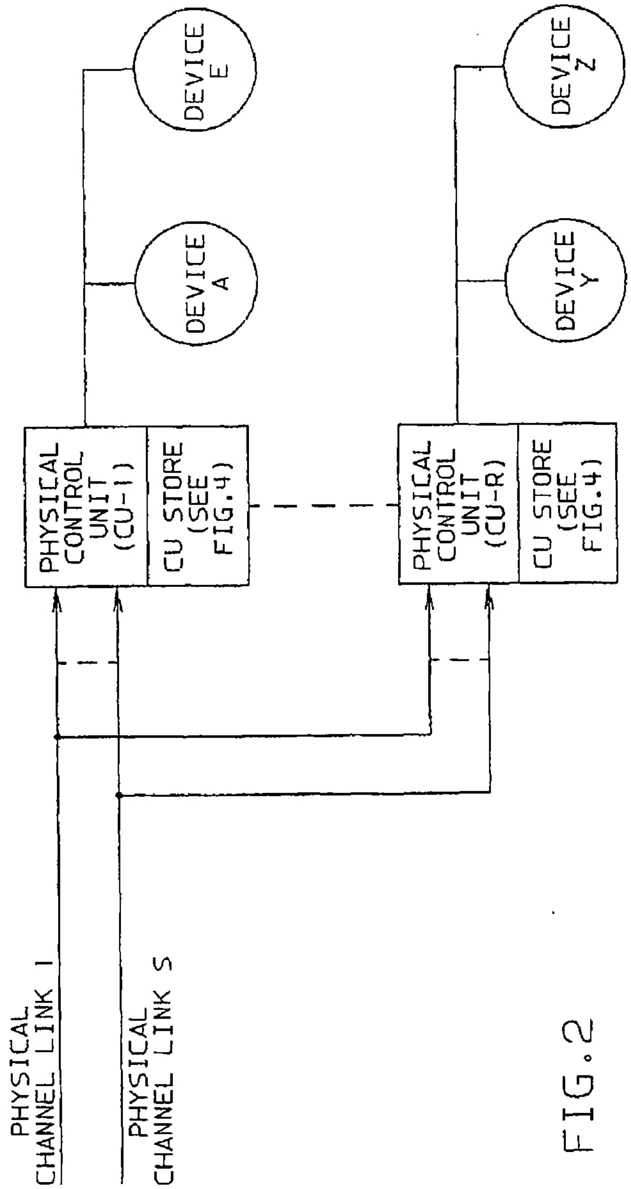

DEVICE E
DEVICE A
DEVICE Z
DEVICE Y
PHYSICAL CONTROL UNIT (CU-1)
CU STORE (SEE FIG.4)
PHYSICAL CONTROL UNIT (CU-R)
CU STORE (SEE FIG.4)
PHYSICAL CHANNEL LINK 1
PHYSICAL CHANNEL LINK S

FIG.2

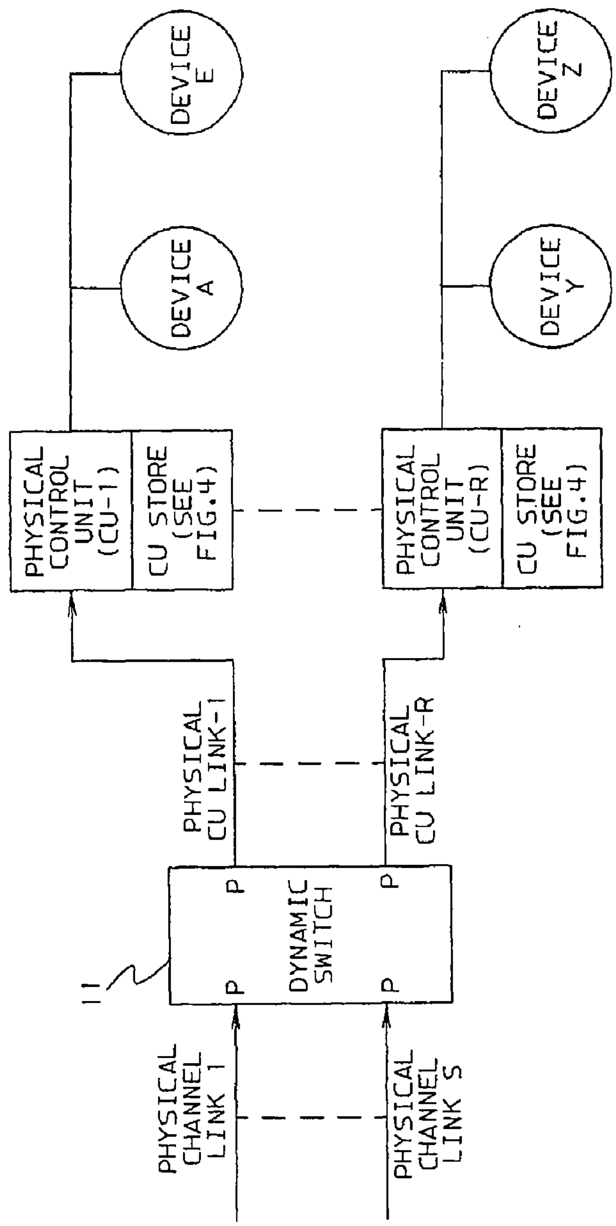

DEVICE E
DEVICE A
DEVICE Z
DEVICE Y
PHYSICAL CONTROL UNIT (CU-1)
CU STORE (SEE FIG.4)
PHYSICAL CONTROL UNIT (CU-R)
CU STORE (SEE FIG.4)
PHYSICAL CU LINK-1
PHYSICAL CU LINK-R
11
P
P
P
P
DYNAMIC SWITCH
PHYSICAL CHANNEL LINK 1
PHYSICAL CHANNEL LINK S

FIG. 3

FIG. 4

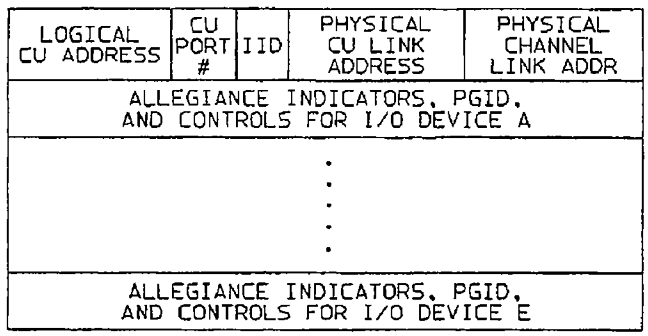

LOGICAL CU ADDRESS
CU PORT #
IID
PHYSICAL CU LINK ADDRESS
PHYSICAL CHANNEL LINK ADDR
ALLEGIANCE INDICATORS, PGID, AND CONTROLS FOR I/O DEVICE A
ALLEGIANCE INDICATORS, PGID, AND CONTROLS FOR I/O DEVICE E

FIG. 5

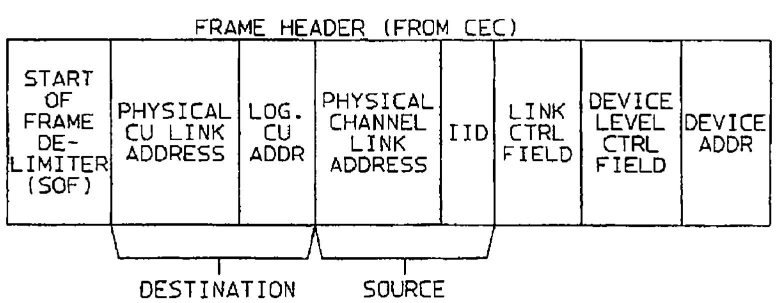

FRAME HEADER (FROM CEC)
START OF FRAME DE-LIMITER (SOF)
PHYSICAL CU LINK ADDRESS
LOG. CU ADDR
PHYSICAL CHANNEL LINK ADDRESS
IID
LINK CTRL FIELD
DEVICE LEVEL CTRL FIELD
DEVICE ADDR
DESTINATION
SOURCE

FIG. 6

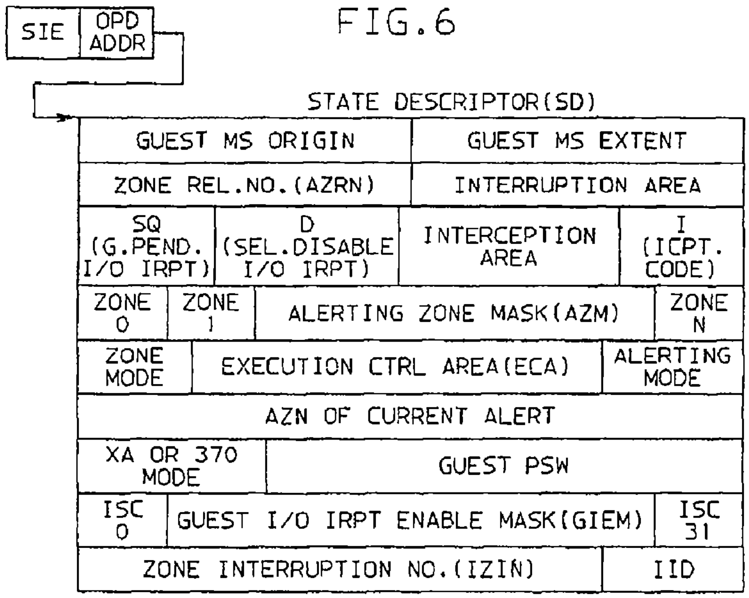

SIE
OPD ADDR
STATE DESCRIPTOR(SD)
GUEST MS ORIGIN
GUEST MS EXTENT
ZONE REL.NO.(AZRN)
INTERRUPTION AREA
SQ (G.PEND. I/O IRPT)
D (SEL.DISABLE I/O IRPT)
INTERCEPTION AREA
I (ICPT. CODE)
ZONE 0
ZONE 1
ALERTING ZONE MASK(AZM)
ZONE N
ZONE MODE
EXECUTION CTRL AREA(ECA)
ALERTING MODE
AZN OF CURRENT ALERT
XA OR 370 MODE
GUEST PSW
ISC 0
GUEST I/O IRPT ENABLE MASK(GIEM)
ISC 31
ZONE INTERRUPTION NO.(IZIN)
IID

FIG. 7

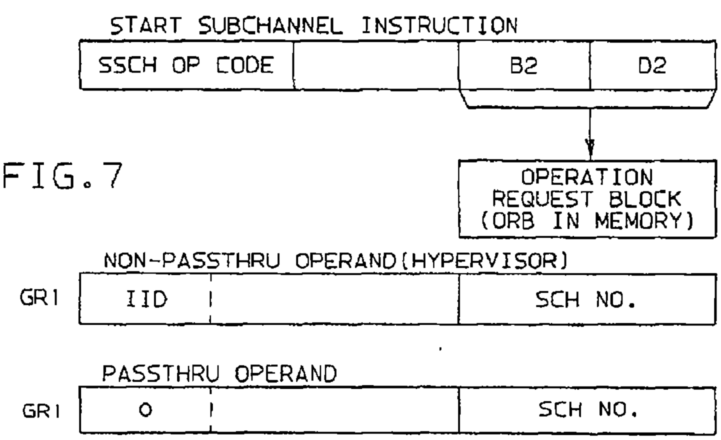

START SUBCHANNEL INSTRUCTION
SSCH OP CODE
B2
D2
OPERATION REQUEST BLOCK (ORB IN MEMORY)
NON-PASSTHRU OPERAND(HYPERVISOR)
GR1
IID
SCH NO.
PASSTHRU OPERAND
GR1
0
SCH NO.

FIG. 8

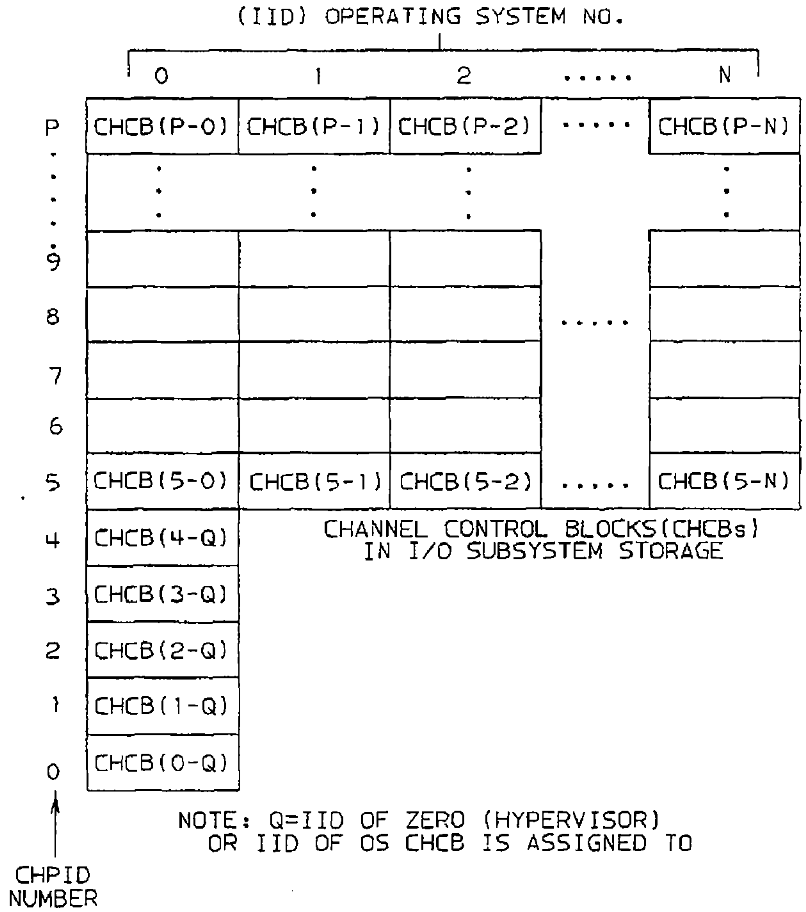

(IID) OPERATING SYSTEM NO.
0
1
2
N
P
9
8
7
6
5
4
3
2
1
0
CHCB(P-0)
CHCB(P-1)
CHCB(P-2)
CHCB(P-N)
CHCB(5-0)
CHCB(5-1)
CHCB(5-2)
CHCB(5-N)
CHCB(4-Q)
CHCB(3-Q)
CHCB(2-Q)
CHCB(1-Q)
CHCB(0-Q)
CHANNEL CONTROL BLOCKS(CHCBs)
IN I/O SUBSYSTEM STORAGE
NOTE: Q=IID OF ZERO (HYPERVISOR)
OR IID OF OS CHCB IS ASSIGNED TO
CHPID
NUMBER

FIG. 9

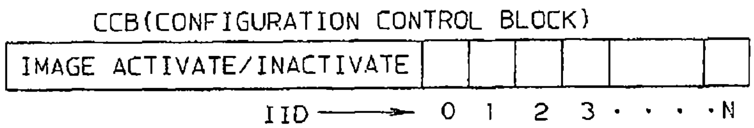
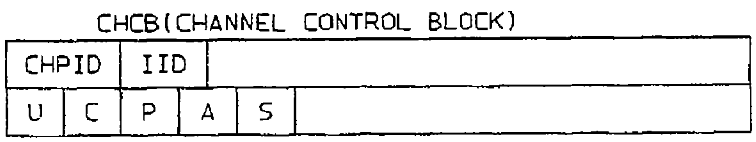


FIG. 10



U=UNSHARED/SHARED INDICATION
C=VARIED ONLINE/OFFLINE INDICATION
P=PERMANENT ERROR INDICATION
A=CANDIDATE INDICATION
S=SUPPRESSED INDICATION
NOTE: IID & CHPID LOCATE CHCB

FIG. 11

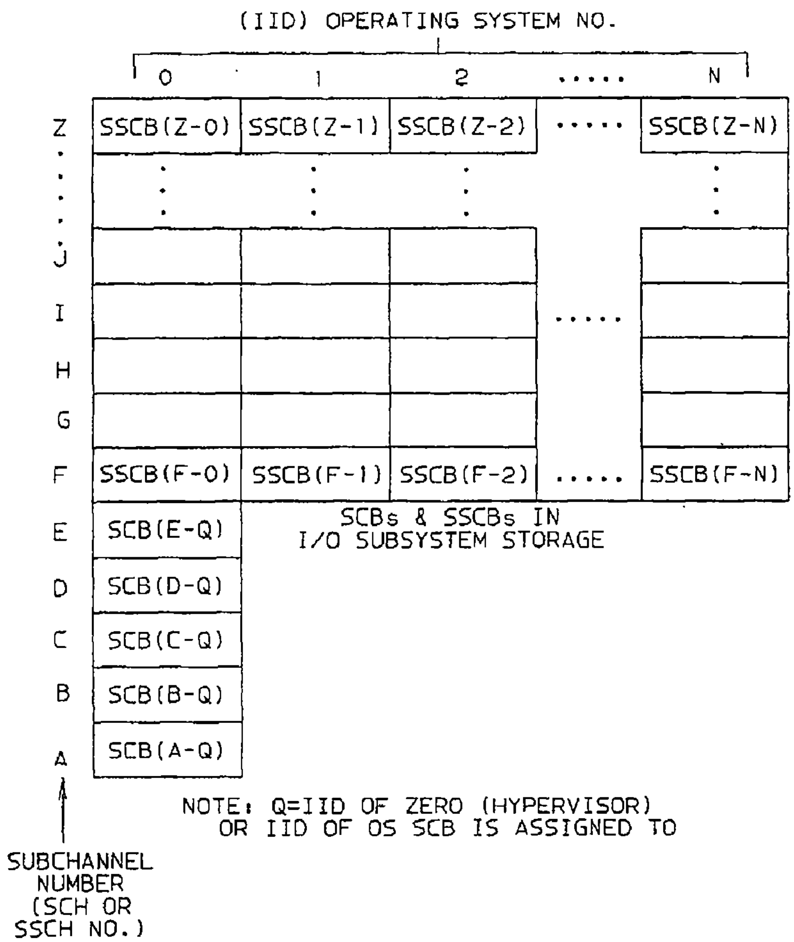

(IID) OPERATING SYSTEM NO.
0
1
2
N
Z
J
I
H
G
F
E
D
C
B
A
SSCB(Z-0)
SSCB(Z-1)
SSCB(Z-2)
SSCB(Z-N)
SSCB(F-0)
SSCB(F-1)
SSCB(F-2)
SSCB(F-N)
SCB(E-Q)
SCB(D-Q)
SCB(C-Q)
SCB(B-Q)
SCB(A-Q)
SCBs & SSCBs IN I/O SUBSYSTEM STORAGE
NOTE: Q=IID OF ZERO (HYPERVISOR) OR IID OF OS SCB IS ASSIGNED TO
SUBCHANNEL NUMBER (SCH OR SSCH NO.)

FIG. 12

SHARED SUBCHANNEL CONTROL BLOCK (SSCB)

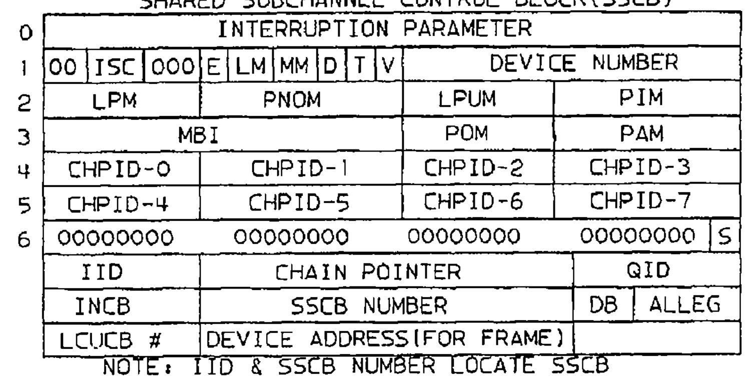


FIG. 14

LOGICAL CONTROL UNIT CONTROL BLOCK (LCUCB)

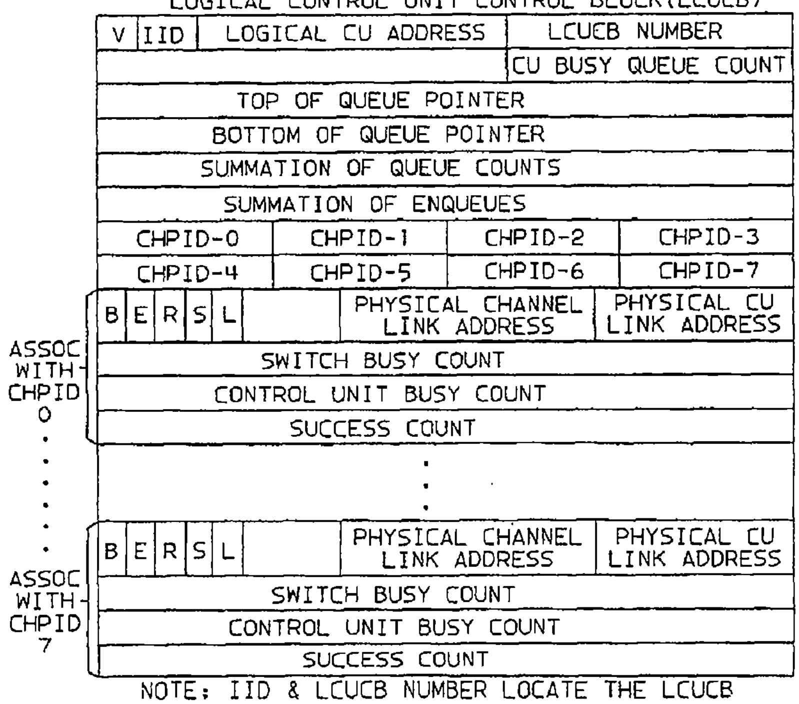


FIG. 13

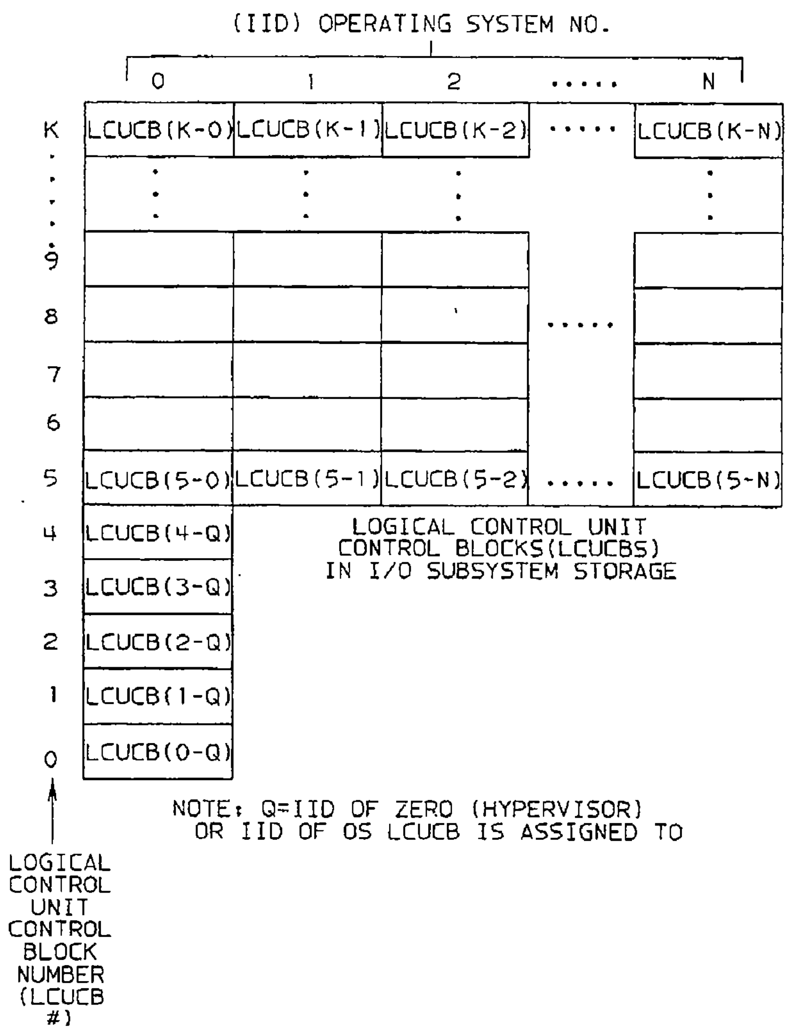

(IID) OPERATING SYSTEM NO.
0
1
2
.....
N
K
9
8
7
6
5
4
3
2
1
0
LCUCB(K-0)
LCUCB(K-1)
LCUCB(K-2)
LCUCB(K-N)
LCUCB(5-0)
LCUCB(5-1)
LCUCB(5-2)
LCUCB(5-N)
LCUCB(4-Q)
LCUCB(3-Q)
LCUCB(2-Q)
LCUCB(1-Q)
LCUCB(0-Q)
LOGICAL CONTROL UNIT CONTROL BLOCKS(LCUCBS) IN I/O SUBSYSTEM STORAGE
NOTE: Q=IID OF ZERO (HYPERVISOR) OR IID OF OS LCUCB IS ASSIGNED TO
LOGICAL CONTROL UNIT CONTROL BLOCK NUMBER (LCUCB #)

FIG.15

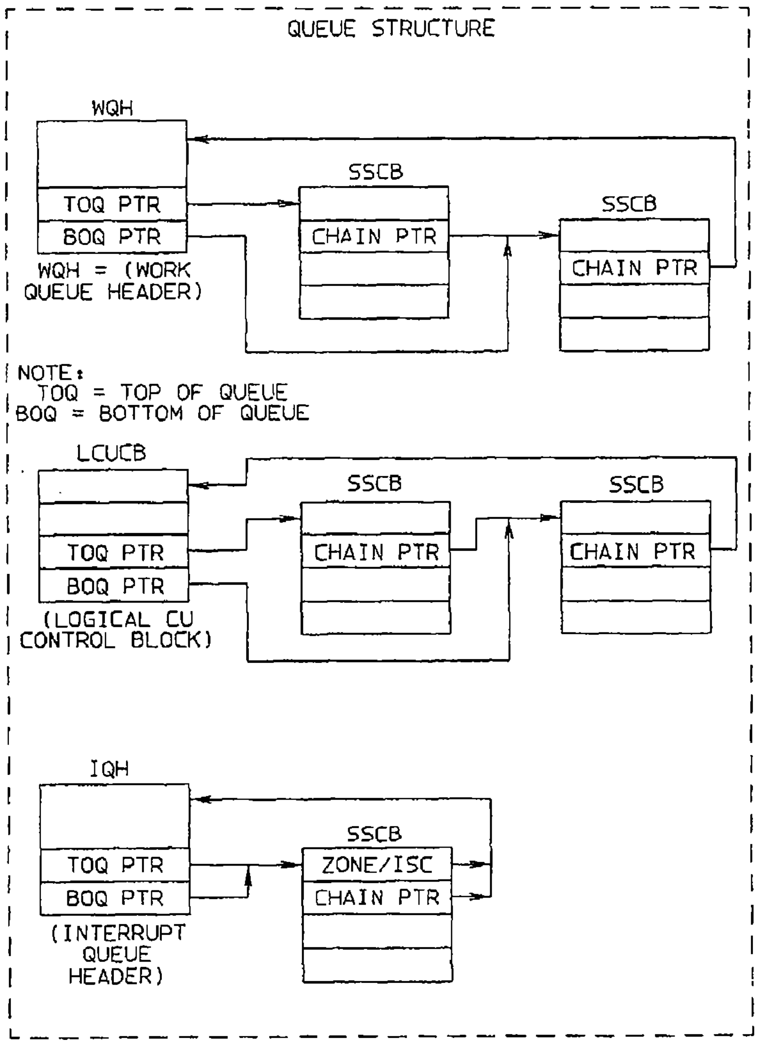

QUEUE STRUCTURE
WQH
TOQ PTR
BOQ PTR
WQH = (WORK QUEUE HEADER)
SSCB
CHAIN PTR
SSCB
CHAIN PTR
NOTE:
TOQ = TOP OF QUEUE
BOQ = BOTTOM OF QUEUE
LCUCB
TOQ PTR
BOQ PTR
(LOGICAL CU CONTROL BLOCK)
SSCB
CHAIN PTR
SSCB
CHAIN PTR
IQH
TOQ PTR
BOQ PTR
(INTERRUPT QUEUE HEADER)
SSCB
ZONE/ISC
CHAIN PTR

FIG. 16A

I/O SUBSYSTEM OF A DATA PROCESSING SYSTEM

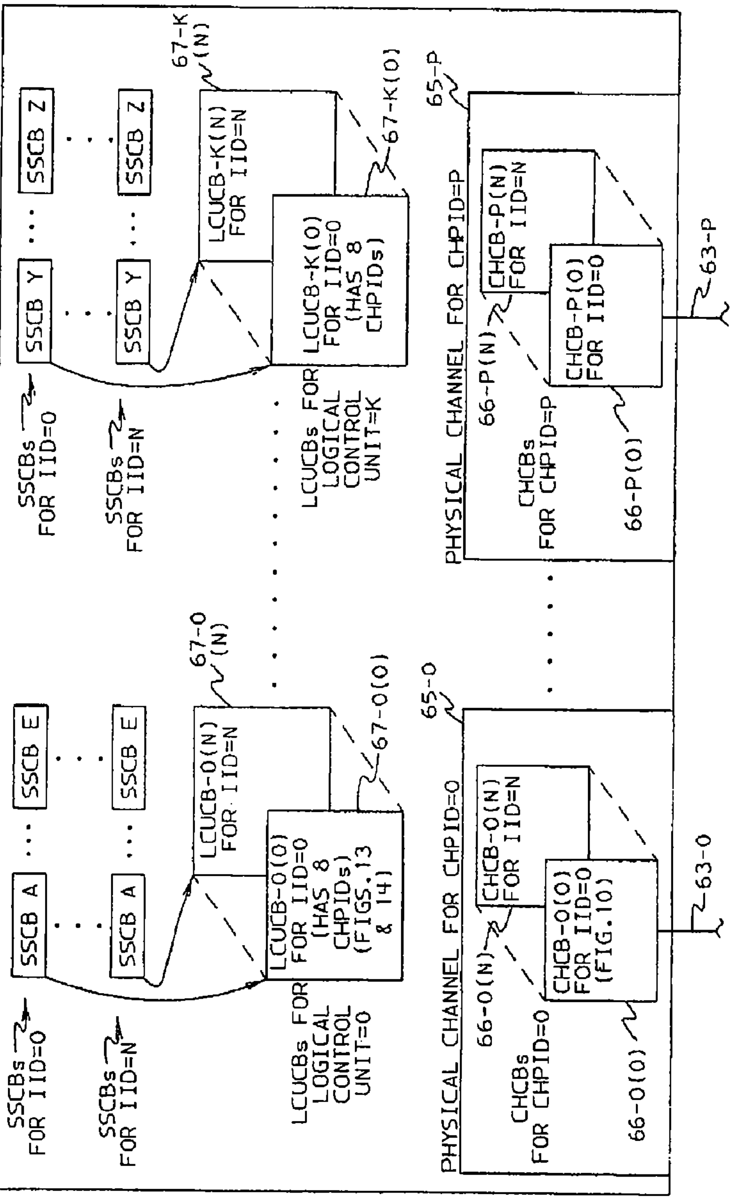
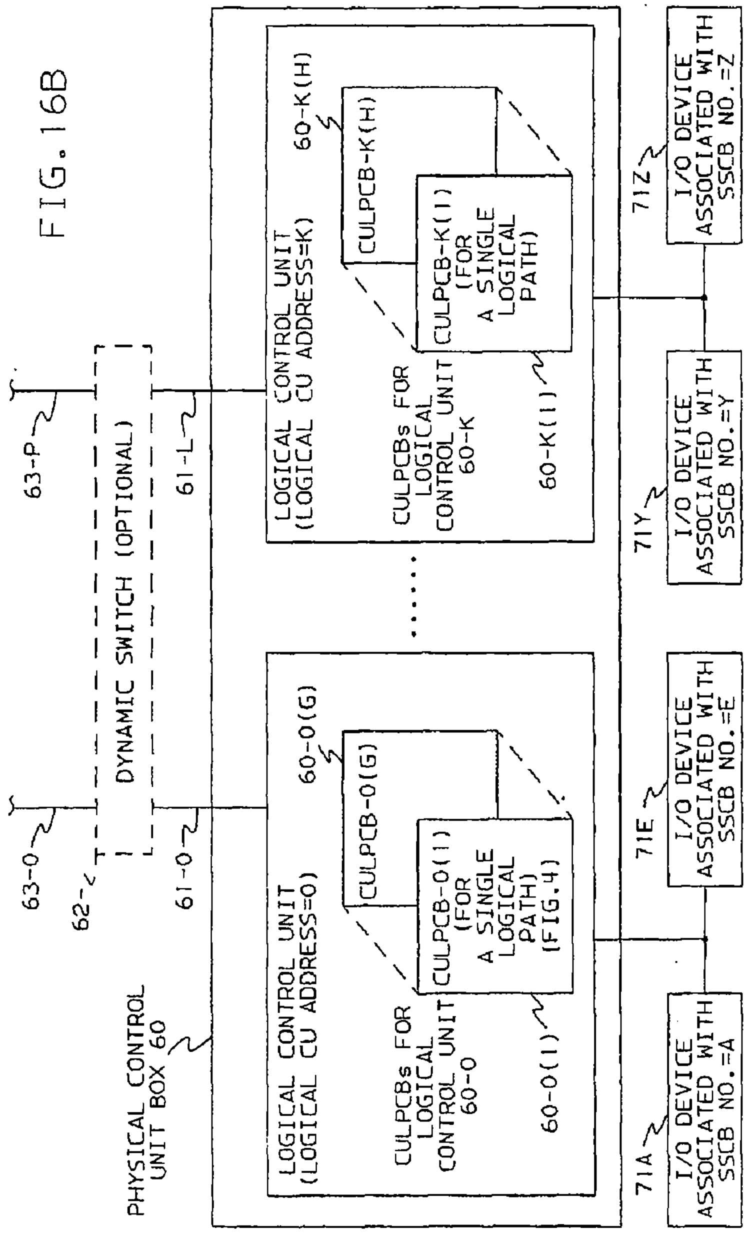


FIG. 16B



63-P
63-0
62
DYNAMIC SWITCH (OPTIONAL)
61-L
61-0
PHYSICAL CONTROL UNIT BOX 60
LOGICAL CONTROL UNIT (LOGICAL CU ADDRESS=K)
60-K(H)
CULPCB-K(H)
CULPCB-K(I) (FOR A SINGLE LOGICAL PATH)
CULPCBs FOR LOGICAL CONTROL UNIT 60-K
60-K(I)
LOGICAL CONTROL UNIT (LOGICAL CU ADDRESS=0)
60-0(G)
CULPCB-0(G)
CULPCB-0(I) (FOR A SINGLE LOGICAL PATH) (FIG.4)
CULPCBs FOR LOGICAL CONTROL UNIT 60-0
60-0(I)
71Z
I/O DEVICE ASSOCIATED WITH SSCB NO.=Z
71Y
I/O DEVICE ASSOCIATED WITH SSCB NO.=Y
71E
I/O DEVICE ASSOCIATED WITH SSCB NO.=E
71A
I/O DEVICE ASSOCIATED WITH SSCB NO.=A

FIG. 17

```
                  101:SIE SD PREVIOUSLY
                  LOADED INTO MEMORY FOR
                     FOR PARTITION J
                            |
                            V
               102:OS PREVIOUSLY LOADED
                      IN PARTITION J
                            |
                            V
         103:HYPERVISOR DISPATCHES OS ON A CPU
                            |
                            V
      104:OS DISPATCHES APPLICATION PROGRAM AS TASK
                            |
                            V
                  105:TASK REQUESTS I/O
            VIA OS INTERFACE SUCH AS READ, GET,
                      WRITE, PUT, ETC.
                            |
                            V
     106:OS IOS ISSUES SSCH INSTRUCTION TO START I/O DEVICE
                            |
                            V
         107:CPU OBTAINS IID FROM SIE SD, AND
            SUBCHANNEL NUMBER IS OBTAINED FROM GR1
            (IID & SUBCHANNEL NUMBER COLLECTIVELY
         LOCATE SSCB FOR THE PROPER SUBCHANNEL IMAGE)
                            |
                            V              N
          108:IS LOCATED SSCB VALID ? ----> 109:HYPERVISOR INTERCEPTS
                            | Y                  SIE INSTRUCTION
                            |
                            V                  N
     111:IS SSCB ENABLED FOR I/O PASSTHRU ? ---> 112:SELECT SSCB WITH
                            | Y                    IID=0 OF HYPERVISOR
                            V                             |
  115:CPU EXECUTES SSCH WITH LOCATED SSCB                 V
        USING PASSTHRU UNDER SIE              113:HYPERVISOR INTERCEPTS
                            |                      SIE INSTRUCTION
                            V
  116:SSCH EXECUTED SUCCESSFULLY ? ---------> 117:EXCEPTION
                            |Y          N                 |
                            V                             |
  119:CPU ENQUES SSCB ON I/O WORK QUEUE OF IOP            V
                            |                    118:HYPERVISOR
                            V                      INTERCEPTION
  120:IOP DEQUES SSCB AND SELECTS A CHANNEL PATH
     (INCLUDING SELECTING AN AVAILABLE CHANNEL
      & ASSOCIATED PATH TO THE REQUESTED DEVICE)
                            |
                            V
  121:IOP ISSUES COMMAND TO SELECTED CHANNEL FOR
          ACCESSING REQUESTED DEVICE USING
          IID, CHPID, AND SUBCHANNEL NUMBER
                            |
                            V
                   (GO TO FIGURE 18)
```

FIG. 18

```
                  (FROM FIGURE 17)
                         |
                         V
131:CHANNEL PROCESSOR LOCATES SSCB USING
        IID AND SUBCHANNEL NUMBER
                         |
 132:                    V
 CHANNEL PROCESSOR CONSTRUCTS TRANSMISSION FRAME CONTAINING:
   IID (I.E. SOURCE LOGICAL ADDRESS),
   PHYSICAL CHANNEL LINK ADDRESS (I.E. SOURCE LINK ADDRESS),
   LOGICAL CU ADDRESS (I.E. DESTINATION LOGICAL ADDRESS),
   PHYSICAL CU LINK ADDRESS (I.E. DESTINATION LINK ADDRESS),
   AND DEVICE ADDRESS
                         |
                         V
 133:TRANSMIT FRAME FROM CHANNEL TO ADDRESSED CU
                         |
                         V
134:CU ACCESSES DEVICE ADDRESSED BY FRAME
        & ACCESSES REQUESTED DATA
                         |
136:                     V
CU CONSTRUCTS RETURN FRAME CONTAINING:
   LOGICAL CU ADDRESS (I.E. SOURCE LOGICAL ADDRESS),
   PHYSICAL CU LINK ADDRESS (I.E. SOURCE LINK ADDRESS),
   IID (I.E. DESTINATION LOGICAL ADDRESS),
   PHYSICAL CHANNEL LINK ADDRESS (I.E. DESTINATION LINK ADDRESS),
   AND DEVICE ADDRESS
                         |
                         V
  137:CHANNEL PROCESSOR RECEIVES FRAME &
      LOCATES SCB USING WITH RECEIVED IID,
   LOGICAL CU ADDRESS, PHYSICAL CU LINK ADDRESS,
           AND DEVICE ADDRESS
                         |
                         V
138:CHANNEL PROCESSOR PLACES ENDING STATUS INTO SSCB
                         |
                         V
139:CHANNEL PROCESSOR SIGNALS I/O IS COMPLETE TO IOP
                         |
                         V
 141:IOP ENQUES SSCB ON I/O INTERRUPT QUEUE FOR OS(IID)
                         |
                         V
  143:FIRST CPU FOR OS(IID) ENABLED FOR INTERRUPTIONS
       HANDLES THE I/O INTERRUPTION &
 CONSTRUCTS AN I/O INTERRUPT RESPONSE BLOCK (WITHOUT IID)
                         |
                         |
                         V
                        END
```

FIG. 19

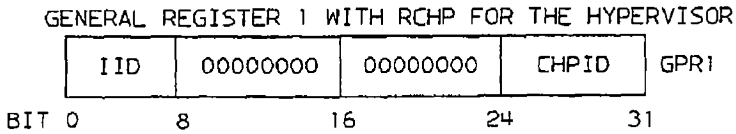
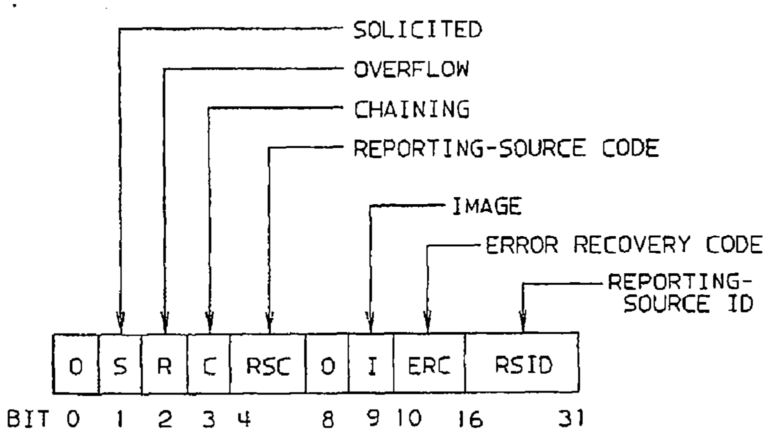


FIG. 20



CRW PRESENTED TO THE HYPERVISOR

METHOD AND MEANS FOR SHARING I/O RESOURCES BY A PLURALITY OF OPERATING SYSTEMS

INCORPORATION BY REFERENCE

The entire contents of the following USA patent applications, filed on the same day as the subject application, are incorporated by reference into this specification: application Ser. No. 07,898,623, now U.S. Pat. No. 5,265,240, (PO9-92-026), entitled "Channel Measurement Method And Means" by R. E. Gailbraith et al; application Ser. No. 07/898,977 (PO9-92-028), entitled; "Asynchronous Command Support For Shared Channels For A Computer Complex Having Multiple Operating Systems" by M. P. Brown et al; application Ser. No. 07/898,875 (PO9-92-029), entitled "Pass-Thru For I/O Channel Subsystem Call Instructions for Accessing Shared Resources In A Computer System Having A Plurality Of Operating Systems" by K. J. Fredericks et al.

Also the following prior-filed applications assigned to the same assignee as the subject application have their entire content incorporated by reference into this specification: application Ser. No. 07/444,190, filed Nov. 28, 1989, by C. J. Bailey et al, entitled "Method And Apparatus For Dynamically Managing I/O Connectivity" (Docket Number KI989013); application Ser. No. 07/754,813, filed Sep. 4, 1991, by R. Cwiakala et al, entitled "Establishing Synchronization Of Hardware And Software I/O Configuration Definitions" (Docket Number PO991036); application Ser. No. 07/676,603, filed Mar. 28, 1991, by S. M. Benson et al, entitled "Method And Apparatus For Dynamic Changes To System I/O Configuration" (Docket Number PO990026); application Ser. No. 07/755,246, filed Sep. 5, 1991, by J. E. Bostick et al, entitled "Method And Apparatus For Dynamically Changing The Configuration Of A Logically Partitioned Data Processing System" (Docket number PO991028); application Ser. No. 07/693,997, filed Mar. 28, 1991, by R. Cwiakala et al, entitled "Dynamically Changing A System I/O Configuration Definition" (Docket Number PO991012); application Ser. No. 07/860,797 filed Mar. 30, 1992 by J. A. Frey et al, entitled "Management of Data Objects Used to Maintain State Information for Shared Data Objects" (Docket Number PO992004); and application. Ser. No. 07/860,646, filed Mar. 30, 1992 by D. A. Elko et al, entitled "Message Path Mechanism for Managing Connections Between Processors and a Coupling Facility" (Docket Number PO992006).

INTRODUCTION

This invention provides a method that greatly increases the effective number of I/O channels, devices and control unit images available to each of a plurality of operating systems (OSs) running on a CEC (computer electronic complex) without increasing the actual number of physical I/O resources. The invention enables the OSs to directly share physical I/O resources without intervention from a hypervisor.

BACKGROUND

In the prior art, either only physical I/O channel resources or only I/O device resources, but not both, were directly sharable by operating systems (OSs) executing in different logical resource partitions of a CEC (computer electronic complex) system. The OSs in a CEC are coordinated by a hypervisor, in which the processor and memory resources of the CEC have been divided among the separately executing OSs. The hypervisor may be structured in internal code (e.g. microcode) or in software. An example of an internal code type of hypervisor is the IBM PR/SM (processor resource/system manager), which coordinates resource contentions among independently executing OSs in separate logical resource partitions. An example of a software hypervisor is the IBM S/370 VM/MPG (virtual machine/multiple preferred guests) system, in which so-called virtual machines (called preferred guests) execute separate OSs in respective logical resource partitions divided by the system software in a software directory.

In prior systems, an I/O channel could be directly shared only by assigning each OS which shared the channel a mutually exclusive subset of I/O devices which could be accessed via that channel. When this technique was used, a single subchannel existed to represent each I/O device, and this subchannel was assigned to the OS which was assigned the corresponding device. Because it is often desired to have I/O devices shared by plural OSs, this technique was very limiting.

In prior systems, an I/O device could be directly shared only by assigning each OS which shared the device a mutually exclusive subset of I/O channels which could be used to access that device. When this technique was used, a plurality of subchannels existed to represent each I/O device, and one of these subchannels were assigned to each OS which shared the device. Each subchannel representing the same device was identified by a different subchannel number. Because each I/O channel was assigned to a single OS with this technique, the number of channels needed would usually increase with the number OSs which were to share I/O devices. This commonly presented a problem since the quantity of channels was limited to 256 due to the 8-bit number which was used to identify them. The quantity of subchannels was less of a problem since the quantity of subchannels had a higher limit of 65536 due to the 16-bit number which was used to identify them.

It was possible in prior systems to share, but not directly share, both I/O devices and the I/O channels used to access these devices. However, this involved a large amount of inefficient system overhead due to the need to intercept to the hypervisor code for each I/O operation in order that the hypervisor could coordinate resource contentions. While the hypervisor code was executing on behalf of the OS, the OS was suspended.

In practice, the overhead of using the hypervisor to obtain sharing of both I/O devices and the I/O channels used to access these devices was so inefficient that most often the choice was made to directly share either only I/O channels or only I/O devices. This allowed all I/O operations for the OS to be performed without hypervisor involvement. This direct use of I/O resources by an OS is called "I/O passthru" because these I/O operations "passthru" (i.e. bypass) the hypervisor.

In the prior art, a System/390 (S/390) CEC has an I/O channel subsystem having one or more I/O processors (IOPs) to control a plurality of I/O channel processors (CHPRs) in the CEC for controlling a like number of channels, which may be fiber optic channels or parallel wire channels connecting to I/O control units with I/O devices. These are the channels involved in the previously discussed hypervisor and OS control. A

widely used type of fiber optic channel uses the IBM ESCON architecture. The CEC consists of one or more central processors (CPUs), system memory, and the I/O subsystem. All of these parts of a CEC are included in the CEC resources used by programs executing in the CEC.

A control unit is the conduit for the exchange of information between an I/O device and a channel. Similarly a channel is the operating system's conduit for the exchange of information between main storage and an I/O device.

An IBM publication (form number SA22-7202) published October 1990 entitled "ES Architecture 390 ESCON I/O Interface" describes the then existing ESCON channel/control unit path connections.

The various resources in the CEC are divided among the OSs by using a plurality of directories or state descriptors (SDs) in system memory that respectively assign the CEC resources to the OSs executing in the respective resource partitions. The CEC hypervisor may be allocated its own logical resource partition to control the overall operation of the CEC, including the dispatching of OSs on the central processors (CPUs) in the CEC, and resolving conflicts among the OSs. Each OS controls the dispatching of application programs running under the respective OS, usually without hypervisor involvement (unless an exception occurs).

Early hypervisor systems required the hypervisor to control all I/O operations for all OSs in the CEC (e.g. early VM/370), including having the hypervisor assign all channel operations, start all subchannels for all I/O devices in the CEC, and handle all I/O interruptions from the devices for all programs running under the OSs.

U.S. Pat. No. 4,843,541 (PO9-87-002) entitled "Logical Resource Partitioning of a Data Processing System", assigned to the same assignee as the subject application, describes and claims a system having "I/O passthru" to enable each OS in a CEC to handle its own I/O operations using dedicated I/O channels and devices without involving the hypervisor. This passthru (or passthrough) feature allowed each OS to start I/O operations requested by application programs running under the OS, and to handle the I/O interruptions resulting from such I/O start operations. The hypervisor only needed to intercept an OS I/O operation when an exception condition occurred. That invention is used in the IBM PR/SM LPAR and S/370 VM MPG systems.

U.S. patent application Ser. No. 07/752,149 (PO9-91-035) filed on Aug. 29, 1991, entitled "CPU Expansive Gradation of I/O Interruption Subclass Recognition", assigned to the same assignee as the subject specification, enables a significant increase in the number of logical resource partitions and CPUs in a CEC. This application enabled each of the CPUs in a CEC (executing any OS running in the CEC) to handle all of the I/O interruption subclasses available in the system; this avoided a prior constraint that restricted each OS to only handling interruptions for one of the I/O interruption subclasses available in the system.

A subchannel is specified for each I/O device supported by an OS under the IBM S/390 architecture. A SCHIB (subchannel information control block), stored in system memory when the S/390 Store Subchannel Instruction (STSCH) is executed, is the means for an OS to view its resources for a subchannel, including the set of channels usable by the subchannel. Each SCHIB contains fields for up to eight channel identifiers, called channel path identifier (CHPID) values, each of which specifies a channel which can be selected for use by the subchannel. An available one of the specified CHPIDs is selected for each data transmission request of the subchannel which is not busy at the time of a subchannel request. In prior systems where I/O devices were directly shared but each channel used to access these devices were assigned to a single OS, the SCHIB could only specify a channel as available when that channel was assigned to the OS.

In prior S/370 and S/390 computer systems, each channel was represented by a single "channel control block" (CHCB) in a CEC's I/O subsystem storage. And each subchannel was also represented by a single subchannel control block (SCB) in the CEC's I/O subsystem storage. An SCB was used by the I/O subsystem internal code (microcode) to select one of up to eight channels specified for the SCB for accessing the I/O device represented by the SCB (the channel assignments of the SCB were the same as in a corresponding SCHIB). Each SCB was assigned to one and only one OS in the CEC. Therefore the assigned OS was the only OS which could access the subchannel corresponding to the SCB using passthru to improve system efficiency (by avoiding hypervisor intervention in managing the I/O operation). No other OS could directly use the subchannel.

In prior systems where I/O devices were directly shared but each channel used to access these devices were assigned (dedicated) to a single OS, an adverse consequence was that when a dedicated channel was utilized only a small percentage of time by its assigned OS, the channel could not be dynamically switched to another OS using passthru; only non-passthru hypervisor accessing (non-direct sharing) was available with its resulting inefficiencies. Consequently, dedicated channels generally remained under-utilized. (The available manual switching of a channel to a different OS did not permit dynamic online switching of an I/O channel to another OS.)

Limiting the number of channels to each OS had the effect of limiting the I/O data rate available to the OS by restricting the number of simultaneous parallel paths for data transmission.

Prior to invention of logical channel paths for the IBM ESCON I/O Interface architecture, a physical relationship existed between either a System/370 or 370-XA channel and an attached I/O control unit and its associated I/O devices. That is, a plurality of physical ports on a control unit were respectively connected to different channels, associating a different channel with each port. Each channel associated with a port was assumed by the control unit to be used by a different OS unless a special command was received from two or more of these channels binding them into a channel path group for the same OS. The channel path group included channel paths connecting the same operating system to the plurality of ports of a CU, and the group was assigned a path group identifier (PGID).

Dynamic switching between channels and control units U.S. Pat. No. 5,107,489 (PO988011), issued Apr. 21, 1992, entitled "Switch And Its Protocol For Making Dynamic Connections", was provided by the ESCON I/O Interface architecture. Dynamic switching allowed a plurality of channels to connect to a single port on a control unit, instead of each channel requiring a connection to a different port. These channels could be on the same CEC or different CECs. Dynamic switching also

allowed a plurality of control unit ports to connect to a single channel.

U.S. patent application Ser. No. 07/576,561, filed Aug. 31, 1990, entitled "Logical Channel Paths In A Computer I/O System" (Docket Number PO990015), assigned to the same assignee as the subject application, describes the invention of logical channel paths. The ESCON I/O Interface architecture eliminated the prior requirement for a channel-to-port connection by the invention of logical channel paths. The concept of logical channel paths made it possible for the control unit to uniquely recognize any of the plural channels to which one of its ports could be dynamically connected. It also made it possible for the channel to uniquely recognize any of the plural control unit ports to which it could be dynamically connected. The control unit continued to assume that each channel capable of connecting to one of its ports was to be used by a different OS unless a special command was received from two or more of these channels binding them into a channel path group for the same OS. The channel path group included logical channel paths connecting the same operating system to the plurality of ports of a CU, and the group was assigned a path group identifier (PGID).

With logical channel paths, each channel and control unit port is assigned a link address. For each channel capable of being connected to a particular control unit port, a unique identifier (physical channel link address) is assigned, which when passed to a control unit port uniquely identifies the channel with respect to that control unit port. For each control unit port capable of being connected to a particular channel, a unique identifier (physical CU link address) is assigned, which when passed to a channel uniquely identifies the control unit port with respect to that channel.

The ESCON I/O Interface architecture also provided for the capability to have a plurality of logical control units exist within a physical control unit. The ESCON I/O architecture calls these logical control units "control unit images", however, herein they are called "logical control units". A logical control unit provides the functions and has the logical appearance of a control unit. When plural logical control units do not exist within a physical control unit, a single logical control unit is said to exist in the physical control unit. Connections between a particular channel and control unit port could be used for some or all of the logical control units which existed in a physical control unit. In order to identify the logical control unit within a physical control unit, a unique identifier (logical CU address) was assigned to each logical control unit.

In each frame header sent by a channel to a logical control unit, the channel identified the destination control unit port by including the physical CU link address in the destination link address field of the frame and identified the destination logical control unit by including the logical CU address in the destination logical address field of the frame. The channel also included its physical channel link address in the source link address field of the frame so that the logical control unit could identify the channel which sent the frame. In each frame header sent by a logical control unit to a channel, the logical control unit identified the destination channel by including the physical channel link address in the destination link address field of the frame. The logical control unit also included both the physical CU link address in the source link address field of the frame and the logical CU address in the source logical address field of the frame so that the channel could identify the control unit port and logical control unit which sent the frame.

By placing the proper link and logical addresses in the appropriate source and destination fields of each frame header, the communicating channel and logical control unit are uniquely identified to each other. It was the combination of the physical channel link address, physical CU link address, and logical CU address which uniquely identified a single logical channel path, with respect to either a physical channel or a control unit port.

Before communication to an I/O device associated with a logical control unit can take place, the establishment of a logical path (LP) is required. The establishing of a logical path is a means for the channel and logical control unit to agree that a particular logical channel path is allowed by both entities to be used for purposes such as transmission of commands, data, and status related to an I/O device. The procedure for establishing a logical path is called the "establish-logical-path procedure". A logical path was uniquely identified by the combination of the physical channel link address, physical CU link address, and logical CU address, with respect to either a physical channel or a control unit port.

SUMMARY OF THE INVENTION

The subject invention significantly increases the number of images of I/O channels, devices (represented by subchannels), and control units directly sharable by a plurality of operating systems (OSs) in a computer electronic complex (CEC) without requiring an increase in the actual number of physical channels, devices or control units connected to the CEC. The OSs can directly share all these physical I/O resources without intervention from a hypervisor.

The subject invention also significantly increases the number of images of I/O channels, devices (represented by subchannels), and control units available to each OS in a multi-OS CEC system without requiring an increase in the actual number of physical channels, physical devices or physical control units connected to the CEC.

A CEC which supports this invention may be said to support a multiple image facility (MIF).

The subject invention may significantly increase the maximum data rate available to each OS in a multi-OS CEC system without requiring an increase in the actual number of channels or devices connected to the CEC. The I/O data rate of an OS is dependent on the parallelism of data transfer to/from the OS. Increasing the number of channels available to each OS can increase the number of I/O devices which can be simultaneously transmitting data to the OS, which can correspondingly increase the maximum data rate available to the OS.

Increasing the parallelism, flexibility and connectivity of channels and I/O devices to each OS (by increasing the number of channels and I/O devices available to the OS) can more quickly obtain different types of data for an OS, even when this does not increase the overall data transmission rate for the OS. The I/O demands of multiple users of an OS are better served by increasing the number of channels and devices connectable to each OS.

Direct control by each of plural OSs over sharable channels and devices by this invention increases system efficiency by avoiding hypervisor intervention. Thus, where previously passthru could not be used for all.

OSs which shared both I/O devices and the I/O channels used to access these devices, this invention is the means which provides this capability.

The great effective increase in I/O channels provided by this invention can easily be expressed using the following example: If a prior CEC had 7 OSs and 84 unshared channels, then each OS had an average of 12 (=84/7) dedicated channels. With this invention, all 84 channels can be made directly sharable by each of the 7 OSs (while still being able to directly share the devices accessible by these channels) so that any OS may now use up to all 84 channels. Accordingly, the number of channels available to any OS has increased from 12 to 84, which is a 700 percent increase in this example.

Of course, any sharable channel may only be active on behalf of one OS at a time, because the channel is usable for an OS only when it is not in a busy state by another OS. In prior CECs, when a channel was dedicated to one OS, it could not be used by any other OS when it was not busy. But with this invention, a sharable channel in a non-busy state can be directly used by another OS, and can be dynamically switched for direct use between different OSs. Thus, a non-busy shared channel may be switched dynamically among plural OSs—whenever needed by any sharing OS. Plural simultaneous requests to a non-busy channel by sharing OSs result in one of the requesting OSs getting the use of the channel and the other requests remaining queued.

Likewise, any sharable device may only be active on behalf of one OS at a time, because the device is usable for an OS only when it is not in a busy state by another OS. In prior CECs, when a device was dedicated to one OS, it could not be used by any other OS when it was not busy. But with this invention, a sharable device in a non-busy state can be directly used by another OS, and can be dynamically switched for direct use between different OSs. Thus, a non-busy shared device may be switched dynamically among plural OSs—whenever needed by any sharing OS. Plural simultaneous requests to a non-busy device by sharing OSs result in one of the requesting OSs getting the use of the device and the other requests remaining queued.

The invention provides a novel method of sharing I/O channels, control units, and devices by a number of different OSs by physically providing multiple control blocks for the respective use by the OSs, each of which specifies a shared resource to an OS, and may be said to represent an image of the resource to each sharing OS. Thus, a sharing set of control blocks for a sharable resource may respectively specify an image of a resource to each sharing OS. A sharing set may represent a single physical channel, a single control unit, or a single subchannel representing a physical I/O device, to plural OSs in a CEC. When plural logical control units exist within a physical control unit, a different sharing set may represent each logical control unit. A sharable channel may access different sharable logical control units and sharable I/O devices. Likewise, a sharable logical control unit may be accessed by different sharable channels.

Each image of each channel, subchannel, or logical control unit is represented in the I/O subsystem by use of a hardware or micro-programming constructs, herein called "channel control blocks" (CHCBs), "sharable subchannel control blocks" (SSCBs), and "logical control unit control blocks" (LCUCBs), respectively. The CHCBs, SSCBs, and LCUCBs are all located in the I/O subsystem storage of the CEC.

All control blocks in a sharing set define the SAME I/O resource. For example, all CHCBs in a sharing set define the same channel to each sharing OS. Each control block in a sharing set is assigned to a different OS by means of a novel "image identifier" (IID). The hypervisor may also be assigned a control block in a sharing set. In the preferred embodiment, IID=0 is assigned to the hypervisor, and the non-zero IIDs are assigned to OSs.

The IID values and the OSs in a CEC need not have a one-to-one correspondence when using this invention. It would be possible for an OS to be assigned more than one IID for its use. But a one-to-one correspondence between an IID value and OS in a CEC is used in the preferred embodiment.

In the preferred embodiment, the IIDs are not visible to the OSs, but are for example, visible to the hypervisor, CPUs, I/O subsystem, and control units.

The IID and the resource number may or may not be designated by fields in each control block of a sharing set, since these values can be implied by the location of the control block in in a two dimensional array in a storage medium. Verification of these values is aided by storing them in respective fields in each control block, and these values are preferably checked in these fields whenever the control block is accessed.

If a sharable resource is selectable by OSs in more than one CEC, then the IID for each OS may be further qualified by storing a CEC identifier along with the IID, e.g. concatenating a unique CEC number with the IID used in the CEC (the IID needing be unique only within its CEC). IIDs need not be unique to a CEC in the preferred embodiment of the invention, however, a unique CEC number is not required due to the logical channel path addressing provided by the ESCON I/O Interface architecture.

The sharable resource identifier may be the resource identifier used in a current architecture, such as the IBM S/390 architecture's use of the "channel path identifier" (CHPID) for channel identification and "subchannel number" for I/O device identification.

A "sharing set" of control blocks used by this invention need not comprehend all OSs represented in the CEC. By not providing a valid control block for an OS in a sharing set, that OS is eliminated from accessing the resource represented by the sharing set, because that OS does not have a valid image of the resource. For example, one or more SSCBs in a sharing set may be missing (or marked invalid) to prevent some of the OSs in the CEC from accessing the device represented by that sharing set. Further, the channel fields in the different SSCBs in the same sharing set need not all specify an identical group of channels, e.g. some channels may be the same and some may be different in the different SSCBs of a set. However in the preferred embodiment of the invention, all OSs are represented in each sharing set for each sharable resource, and the same channel identifiers are specified in all blocks of a sharing set of SSCBs. However, some parameters may differ among the control blocks in a sharing set without spoiling their image capability.

In this invention, non-sharing resource control blocks (like those found in prior systems) may also be intermixed with sharable resources of the same type. Thus, a non-shared subchannel (SCB) may be used for an I/O device dedicated to a single OS, and sharable subchannels (SSCBs) may also be used for enabling passthru I/O operations by plural OSs to that device.

This invention is capable of operating with prior CEC resource partitioning architectures that permit OS programs executing different resource partitions of the CEC, such as in IBM PR/SM system (in which separated resources are defined in different logical partitions), or in the IBM S/370 VM MPG (Virtual Machine Multiple Preferred Guest) system (which uses software directories to defined different logical partitions). Either of these types of partitioned systems can perform I/O operations in a passthru mode which allows OSs to directly share an I/O channel or device (but not both) without any intervention from a CEC hypervisor (if no exception is encountered) to significantly shorten the time for I/O accessing. The I/O channels and devices cannot be both shared for direct accessing by an OS in passthru mode. In these prior systems, all channels and devices can only be accessed by the hypervisor; and hypervisor intervention is needed if OSs are to share both I/O devices and the channels used to access these devices, which is a very inefficient type of I/O operation compared to direct passthru operations by an OS.

The sharable channels used by this invention may provide bit-serial, bit-parallel or serial/parallel types of data transmission. The invention is preferably used with the serial I/O channel interface of the type described by the IBM Enterprise Systems Connection (ESCON) architecture because it has been found simpler to implement; but this invention may be used with other channel architectures. That is, the IBM ESCON I/O Interface architecture provides logical channel paths and logical I/O addressing capabilities for which this invention may be easier to implement.

This invention has found a way to increase the number of channels and subchannels available to each OS in a CEC without changing the size of the channel identifier (CHPID) value or of the subchannel identifier (subchannel number) value. With this invention, the effective number of channels and subchannels available to the OSs in a CEC is a multiple of the number of IIDs activated in the CEC.

And the maximum data rate available to any OS in a CEC is increased by this invention enabling a dynamic shifting on a demand basis in the use of any shared resource (e.g. channels, subchannels and logical control units) to whichever OS in the CEC needs its use. Dynamic shifting on a demand basis significantly improves the utilization of the channels in the CEC.

This invention expands the use of the source and destination address fields of each frame header, as provided by the ESCON I/O Interface architecture, to now include a novel use of the source logical address (for frames sent from a channel to a control unit) and destination logical address (for frames sent from a control unit to a channel). These new logical address fields in the frame header are used to identify either the image of the channel which sent a frame or the image of the channel to which the a frame is being sent. When the channel sends a frame header, it includes the IID for the corresponding channel image in the source logical address field of the frame. This identifies the channel image to the control unit. When a control unit sends a frame header, it includes the IID for the corresponding channel image in the destination logical address field of the frame. This identifies the channel image to the channel.

This invention expands the identification of a logical channel path and logical path (LP) to include the IID corresponding to a channel image. A single logical channel path or logical path is uniquely identified by the combination of the physical channel link address, physical CU link address, IID, and logical CU address, with respect to either a physical channel or a control unit port.

The IID need not be the actual value used in the frame header to identify the channel image. It would be possible to use another identifier which had a one-to-one correspondence with the IID. However, the IID value is included in the frame header in order to identify the channel image in the preferred embodiment.

With this invention, the IID in a frame header can be used for both shared and unshared channels. For unshared channels, a single channel image and a single channel control block are used (for the dedicated channel).

The information in each frame transferred through any physical channel is always restricted to the OS assigned the IID in the frame header. The frame is isolated from all other OSs (using a different IID in their frames), and the IID logic for supporting the images of a channel, subchannel, or logical control unit within a CEC maintains this restriction.

This invention also comprehends use of a dynamic channel switch (called a "director") to support multiple channel images by assigning multiple link addresses to all of the images of the shared channels within the I/O subsystem of a CEC, e.g. one link address per channel image. This is a non-preferred version of this invention for handling the images within a CEC, because the director does not know how many images of a channel a CEC supports or is currently configured with, and so would have to assign the maximum number to each channel, which would deplete the number of link addresses available and produce a more complex director port design in the director. This technique would also require the director to know which ports had channels attached and which of these were shared channels.

With the previously-summarized preferred form of this invention, a single port is required to attach either a shared channel or a unshared channel to a director and the director assigns a single link address to the physical channel. It is the IID included in the frame header which is used to uniquely identify the channel image of a physical channel.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates a computer electronic complex (CEC) using the invention.

FIG. 2 shows physical channel links connecting directly to control units (CU) ports without going through a dynamic switch.

FIG. 3 shows physical channel links connecting to control units (CUs) through a dynamic switch.

FIG. 4 illustrates a "control unit logical path control block" (CULPCB) used in representing a logical path within a logical control unit in a memory used by the control unit.

FIG. 5 represents a frame header (containing logical path identifier components) transmitted by a CEC to an I/O control unit.

FIG. 6 represents a state descriptor (SD) control block of a SIE (start interpretive execution) instruction used for indicating the resources assigned to a partition in a computer electronic complex (CEC).

FIG. 7 represents a start subchannel (SSCH) instruction and its operands for use by an embodiment of the invention.

FIG. 8 is an example of an array containing channel control blocks (CHCBs) used in an embodiment of the invention.

FIG. 9 illustrates an example of the content of a configuration control block (CCB) used to activate the IIDs which may be used by operating systems in a CEC.

FIG. 10 illustrates an example of the content of a channel control block (CHCB) used in an embodiment of the invention.

FIG. 11 is an example of an array containing both non-shared subchannel control blocks (SCB) and shared subchannel control blocks (SSCB) used in an embodiment of the invention.

FIG. 12 illustrates an example of the content of an SSCB or SCB used in an embodiment of the invention.

FIG. 13 is an example of an array containing logical control unit control blocks (LCUCBs) used in an embodiment of the invention.

FIG. 14 illustrates an example of the content of a logical control unit control block (LCUCB) used in an embodiment of the invention.

FIG. 15 represents a working queue (WQH), a control unit image queue (using a LCUCB as a header), and an interruption queue (IQH) used by an embodiment.

FIG. 16A and FIG. 16B together show an integrated embodiment of the invention having different types of control blocks representing various types of IID-associated images for a plurality of operating systems.

FIG. 17 and FIG. 18 provide a flow-diagram of a start subchannel instruction in the preferred embodiment.

FIG. 19 shows the format of General Register 1 provided by this invention.

FIG. 20 shows the format of the channel report word as provided by this invention.

DESCRIPTION OF THE DETAILED EMBODIMENTS

Computer Electronic Complex (CEC)

FIG. 1 shows a computer electronic complex (CEC) used with an embodiment of the invention. The CEC includes one or more central processors (CPUs), a system memory, caches and controls (not shown) of the type found in the prior art for interconnecting the CPUs to the system main memory, and an I/O subsystem. The CEC resources in FIG. 1 are configured into resource partitions 1 through N, which may be done in the manner described and claimed in U.S. Pat. No. 4,848,541 (previously cited herein).

Each of the N partitions in the CEC shown in FIG. 1 contains an operating system (OS), and a microcode hypervisor (such as the IBM PR/SM microcode hypervisor) controls the overall operation of the OSs. Alternatively, the CEC may contain a plurality of OSs that operate under a virtual machine (VM) software hypervisor. In either case, the CEC has N number of OSs simultaneously and independently executing under control of a hypervisor. The OSs may, for example, be copies of the IBM MVS and/or VM CMS systems.

The I/O subsystem in FIG. 1 includes I/O processors (IOPs) 1 through T, and channel processors (CHAN PROCs) 1 through S. The I/O subsystem may, for example, have up to 256 channel processors (when using an eight bit CHPID), and usually has a lessor number of IOPs. The IOPs remove the I/O requests received from the CPUs via an I/O work queue and select a channel processor for controlling the requested I/O operation. The number of IOPs is determined by whatever number can handle the I/O work load from the CPUs in a timely manner. Usually only a few IOPs are required, such as four IOPs, which are presumed in the preferred embodiment. The channel processors respectively control data transmissions on channels 1 through S, each of which may be a serial channel of the IBM S/390 ESCON type in the preferred embodiment.

This invention enables plural OSs simultaneously executing a plurality of I/O channel programs to directly and efficiently share the IOPs and I/O channels.

The sharing of the I/O resources utilizes three types of images of I/O resources, including I/O channel images, logical control unit images, and device images via subchannel images.

Each physical channel is represented by a sharing set of channel control blocks (CHCBs). The CHCBs are located in I/O subsystem storage. The I/O subsystem storage is preferably separate from (for protection from) CPU program addressable storage.

Each OS has a different "channel image" of the same physical channel. The different channel images of the same physical channel are represented by information in each of the CHCBs of the sharing set. Each channel image is defined herein by an OS identifier (IID) and a physical channel identifier (CHPID). The CHCB for a particular channel image can be located in I/O subsystem storage by its associated CHPID and IID values. Various characteristics for each of the channel images for the same physical channel are indicated by settings in the CHCBs for the respective channel images.

An image of the channel is used as a component in defining a logical path (LP) from a particular OS through the associated physical channel to a logical control unit (including through any I/O dynamic switch). A single logical path is uniquely identified by the combination of the physical channel link address, physical CU link address, IID, and logical CU address, with respect to either a physical channel or a control unit port. Different I/O channel programs operating under different OSs may be simultaneously executing by using different images of the same physical channel, although only one channel program can be transmitting commands, data, or status through the physical channel at any one time. FIGS. 8 and 10 show CHCBs, and show how they are organized in an array such that they can be located by CHPID and IID values.

Each subchannel is represented by a sharing set of shared subchannel control blocks (SSCBs). The SSCBs are located in I/O subsystem storage. The I/O subsystem storage is preferably separate from (for protection from) CPU program addressable storage.

Each OS has a different "subchannel image" of the same subchannel. The different subchannel images of the same subchannel are represented by information in each of the SSCBs of the sharing set. Each subchannel image is defined herein by an OS identifier (IID) and a subchannel identifier (subchannel number). The SSCB for a particular subchannel image can be located in I/O subsystem storage by its associated subchannel number and IID values. Various characteristics for each of the subchannel images for the same subchannel are indicated by settings in the SSCBs for the respective subchannel images.

Different I/O channel programs operating under different OSs may be simultaneously executing and sharing the same subchannel (the same device) by using

different images of the same subchannel, although only one channel program can be accessing the device at any one time. FIGS. 11 and 12 show SSCBs.

Each logical control unit is represented by a sharing set of logical control unit control blocks (LCUCBs). The LCUCBs are located in I/O subsystem storage. The I/O subsystem storage is preferably separate from (for protection from) CPU program addressable storage.

Each OS has a different "logical control unit image" of the same logical control unit. The different logical control unit images of the same logical control unit are represented by information in each of the LCUCBs of the sharing set. Each logical control unit image is defined herein by an OS identifier (IID) and a logical control unit identifier (LCUCB number). The LCUCB for a particular logical control unit image can be located in I/O subsystem storage by its associated LCUCB number and IID values. Various characteristics for each of the logical control unit images for the same logical control unit are indicated by settings in the LCUCBs for the respective logical control unit images.

Different I/O channel programs operating under different OSs may be simultaneously executing and sharing the same logical control unit by using different images of the same logical control unit, although only one channel program can be transmitting commands, data, or status through a particular physical channel and control unit port at any one time. FIGS. 13 and 14 show LCUCBs.

Although this invention supports shared channels, shared control units, and shared subchannels, this invention also permits a CEC to have and be using unshared channels, unshared control units, and unshared subchannels (devices) while it is using the shared I/O resources. A particular channel, control unit, or subchannel may be changed from either unshared to shared, or from shared to unshared by dynamically or statically reconfiguring the CEC resource assignments.

The image concept of this invention allows only the OS associated with a particular image resource identifier (IID) to access information acquired with the use of that OS's image identifier. The use of shared resources by this invention need not affect the privacy of I/O information an OS accesses. That is, each OS sharing the same physical resources with other OSs maintains its security of I/O information from all other OSs sharing the resources. And no OS need view its IID or any other IID. In the preferred embodiment, the IIDs are not visible to the OSs, but are for example, visible to the hypervisor, CPUs, I/O subsystem, and control units.

Channel Paths to I/O Devices

FIG. 2 illustrates a set of physical channel links from the CEC in FIG. 1. Concurrently executing channel programs in different OSs in the CEC may be using any single physical channel path to access the same or different I/O devices when using this invention. Any channel path may include elements such as any of the channels 1–S from the CEC to any of control units (CUs) 1 through R. These CUs connect to I/O devices A, E . . . Y, Z. Although channels 1–S are shown respectively connected to S number of ports on each CU, it should be understood that any CU may have anywhere from one to S number of ports. Each of the channels 1–S may be connected to a different port of a CU. And, some channels may not be connected to a particular CU.

FIG. 3 shows the same physical channel links 1–S connected through a dynamic switch 11 to the same control units (CUs) 1–R. The advantage of dynamic switch 11 is to obtain the same connectivity of channels to CUs (as was obtained in FIG. 2 without a dynamic switch), except that in FIG. 3 each CU has only a single port to which any of channels 1–R may be connected. Thus, the dynamic switch 11 eliminates the need for multiple CU ports to obtain flexible channel-to-CU connectivity. The CUs in FIG. 3 are assumed to connect to the same sets of I/O devices A, E . . . Y, Z as in FIG. 2.

FIGS. 2 and 3 are intended to show that the invention comprehends all manners of obtaining physical channel-to-CU paths, whether or not any dynamic switch is provided in the connection path, and whether or not the CUs have one or multiple ports. A dynamic channel switch is sometimes called a "director".

OS Image Identifier (IID)

One or more different "image identifiers" (IIDs) are assigned to each of the plural OSs executing in different resource partitions of the CEC in FIG. 1. In the preferred embodiment herein, one IID is assigned to each OS in a CEC.

Although assigned to OSs, in the preferred embodiment, the IIDs are not visible to the OSs. But for example, they are visible to the hypervisor, CPUs, I/O subsystem, and control units.

The IIDs are used to enable the plural OSs to share the physical I/O resources connectable to the CEC without decreasing the data security between the OSs. The new-found sharability provided by this invention enables up to all of the OSs in a CEC to share up to all of the I/O channels, up to all of the control units (both physical and logical) available to the CEC, and up to all of the I/O devices connected to the control units, without involving the hypervisor in the I/O operations.

The invention enables the OSs to use different images of the same channels, subchannels (representing I/O devices) and logical control units. The different images enable individual sharing and control by each OS of the same physical channel and/or of the same control unit (both physical and logical), and/or of the same physical I/O device.

FIG. 16A represents an example of channel images, logical control unit images, and subchannel images by means of control blocks stored in a CEC's I/O subsystem storage. A control unit also has control blocks stored in I/O control unit storage which represent logical paths, as shown in FIG. 16B. It is the use of these control blocks by the I/O subsystem and the CU which permits the different OSs to access and directly share all the same I/O resources.

Multiple images of the same subchannel permit each OS to be connected (through an OS-related image of the same subchannel) to the same device.

Different OSs sharing a physical channel may asynchronously multiplex data through the same physical channel at different times to the same I/O device or to different I/O devices.

Each of the different channel images is represented by a channel control block (CHCB) in the I/O subsystem storage. The I/O subsystem storage is preferably in a memory area separate from system main storage, so that the I/O subsystem storage is not addressable by CPU instructions, but is addressable by internally coded (microcoded) instructions, and hardware. Examples of

CHCBs are shown in FIGS. 8, 10 and 16A. There may be up to (N+1)*(P+1) of sharable CHCBs in I/O subsystem storage.

The preferred embodiment assigns a unique non-zero IID value to each OS in a CEC, and reserves the IID=0 value for assignment to the CEC hypervisor. The hypervisor may or may not actually be assigned any IID value.

The requirements of different hypervisors vary in relation to whether they need to be able to perform I/O operations on their own behalf. For example, if a software hypervisor is used, it should be able to share I/O devices with its OSs; and then a shared subchannel control block (SSCB) having an IID=0 may be provided in each sharing set of SSCBs for use by the hypervisor. Likewise, a CHCB and LCUCB having an IID=0 may be provided in each sharing set of CHCBs and LCUCBs, respectively. This allows a software hypervisor (such as VM/370 XA) to share I/O channels, CUs and devices with its OSs. On the other hand, if a microcode hypervisor is used (e.g. the hypervisor in the IBM PR/SM LPAR system), it need not share I/O resources with the OSs, in which case no SSCB, CHCB, or LCUCB for the hypervisor (having an IID=0 value). need be provided in each sharing set.

The preferred embodiment makes the IIDs transparent to all OSs in the CEC, and to all programs executing under the OSs. It is not necessary to have any OS, or OS program, be aware that IIDs are being used in the CEC, or that I/O resources are being shared by the OSs. Only the system hypervisor, CPUs, I/O subsystem, and control units need to be aware of IIDs and resource sharing. No OS needs to view or access any IID value, since the OS's IID value is automatically handled by the hypervisor, CPU, I/O subsystem, and control unit controls (including its system microcode and hardware operations) whenever an OS requests an I/O operation. And programs executing under any OS (e.g. in any logical partition or in any virtual machine in a CEC) need not be aware of the existence of IIDs.

IID Value Activation

FIG. 9 illustrates a configuration control block (CCB) in which the various IID numbers are activated or inactivated for use by the CEC operations. A bit position is provided for each possible IID value from zero up to a maximum. A bit position corresponding to a particular IID value is set to a one state to activate that IID value, or is set to a zero state to indicate an inactive state for that IID value.

If each IID is represented by an 8-bit number, it allows up to 255 non-zero values, of which, for example, 63 may be activated and assigned to OSs of a resource-partitioned CEC. The IIDs could be specified by a larger or smaller number, and more than one IID could be assigned to any OS, although in the preferred embodiment there is only one IID assigned to any OS.

No requirement exists for the activated IID values to start at any given value or be in a dense range of IID values. For example, an implementation could provide a set of four active IID values such as 0, 2, 7, and 8 for four associated OSs.

Assignment of an IID to an OS

An IID is assigned to an OS by storing the IID value in a hypervisor control block, called a SD (state description) which is the operand of a SIE (start interpretive execution) instruction, executed only by the hypervisor for dispatching any OS. An exemplary form of an SD is shown in FIG. 6. A respective SD is provided in system main storage for each OS operating under the hypervisor in order to define the subset of CEC resources available to each OS. Accordingly, an IID is assigned to an OS when the assigned IID value is stored in the IID field in the SD assigned to the respective OS. The SDs are not accessible to the OSs in this embodiment.

When an I/O command is issued by any OS, the OS's IID is usually required in the execution of the command. The hypervisor or microcode accesses the IID in the SD, and microcode controlling the execution of the command for the respective OS, applies the IID for the current OS command without the OS having any access to the IID. The microcode locates and accesses any required CHCB, SSCB, and/or LCUCB to select any channel, subchannel, or logical control unit images required for performing the OS command, setting up any required logical path (LP) specification, generating a frame header (containing all addresses required at the CU), and transmitting the frame packet on the selected physical channel path to the CU for accessing a requested I/O device (also addressed in the frame header). The CU stores the communicated LP specification (including its IID) for use by the CU when the CU later must respond to the request (after performing the requested command).

Channel Images (CHCBs)

Images of each channel are provided by channel control blocks (CHCBs) which are used in association with other control blocks. Each physical channel is identified by a respective CHPID number.

Not all of the channels need to be shared, and in FIG. 8, channels corresponding to CHPIDs 0 through 4 are not shared and are assigned to either the hypervisor (IID=0) or one of the OSs (IID other than zero). Any number, including all or none, of the channels may be shared. In FIG. 8, the channels having CHPIDs higher than 4 are shared by all N OSs and CHPIDs 0 through 4 are not shared.

Each CHPID may have a plurality of channel images (CHCBs) up to the number of IIDs (equal to the number of OSs). Each of the channel images independently represents a physical channel to a respective OS, so that the OSs can independently operate the physical channel. That is, each OS which uses a particular physical channel has its own channel image different from the channel images of the other OSs for the same physical channel. Thus, each channel image for any OS may have states different from, and independent of, the channel image of any other OS for the same physical channel.

FIG. 8 shows an example of an array of CHCBs 0(0)-P(N) that represent all channel images for all physical channels in a CEC. The respective CHCBs are indexed (located) in the array by their assigned CHPID, which is written into the first field of each CHCB in the preferred embodiment in which each CHPID is represented by an eight bit number that limits the maximum number of CHPIDs (and the corresponding number of channels in a CEC) to 256. (Note: this example is not the array of CHCBs shown in FIG. 16A in which only CHCBs for shared channels are shown.)

Structure of an CHCB

FIG. 10 shows the content of a CHCB (which is also the content of the CHCBs used in FIG. 16A). The first row in each CHCB contains the value of the CHPID represented by the respective CHCB. An IID field contains the IID assigned to this CHCB. These two values, the CHPID and IID, together locate any CHCB in the I/O subsystem storage, where they are used for accessing any CHCB in the channel operations. Other fields in each CHCB are:

U: Unshare/shared indication indicates if the channel is being unshared (dedicated to one OS), or is shared by a plurality of OSs.

C: Varied online/offline indication which indicates if the respective channel image is varied online where it can be operational, or varied offline where it cannot be operational but where it can be serviced for a maintenance operation.

P: Permanent Error: Indicates whether the channel image is currently in a permanent error condition or not.

A: Candidate: Indicates whether the channel image is permitted to be varied online.

S: Suppressed: Indicates whether new I/O activity may be initiated for the channel image.

There may be other fields (not shown) in each CHCB, in addition to these defined fields which are not unique to the CHCBs in this specification.

The following (and many other) scenarios are possible in the states of these novel channel images for the same or different physical channels:

a) Shared channel images for OSs 1, 2 and 3 are varied online.

b) The channel image for OS 1 is varied offline and is not operational, while the channel images for OS's 2 and 3 are varied online and are concurrently performing I/O operations.

c) A momentary error condition has occurred in the physical channel which causes the channel images for OS's 2 and 3 to be placed in the permanent error state. (The channel image for OS 1 is offline, as varied by the previous step).

d) In order to again use its channel image, OS 2 varies its channel image offline, and then varies the channel image online which cures the error condition and makes the channel image error-free. This results in the three channel images ending up in a different state: The OS 1 channel image is offline; the OS 2 channel image is online and error-free, and the OS 3 channel image is online and in permanent error.

Other Control Blocks per Channel

Other control blocks are used in the operation of the channels, such as for example, a "reverse lookup control block" (RLCB) associated with each CHCB. The RLCB lists each subchannel which can use a respective physical channel.

If there is a channel assignment variation in the subchannel images (SSCBs for the same sharing set), in which some subchannels in the sharing set can use a particular channel and other subchannels in the same sharing set may not, such variation might also be listed in the RLCB. (However, the preferred embodiment herein assigns the same channels (CHPIDs) to all SSCBs in the same sharing set).

Subchannel Images (SCBs and SSCBs)

This invention provides a set of subchannel images for each subchannel by means of a "sharing set" of SCBs, herein called "sharing SCBs" (SSCBs). The SSCBs in a sharing set are respectively assigned to different IIDs (representing different OSs). The novel concept of a sharing set enables up to all OSs in a CEC to access the same device (because the same subchannel number, representing the same device, is assigned to all SSCBs in the same sharing set).

FIG. 11 shows an example having both sharing sets of SSCBs and non-shared SCBs. Each box in FIG. 11 represents an SSCB or an SCB. They are arranged vertically in FIG. 11 according to subchannel numbers A to Z, in which the subchannel numbers correspond to the I/O devices in FIGS. 2 and 3. Subchannel numbers A–E represent only non-shared SCBs. Each of the subchannel numbers F–Z represent a sharing set of SSCBs. As previously stated, each subchannel is assigned to a different I/O device.

The top row in FIG. 11 indicates the OS ownership of the SSCBs. Each column is respectively assigned a different IID from 0 to N to indicate the OS ownership of the SSCBs in the respective column. In column IID=0, the SSCBs belong to the hypervisor, since IID=0 is assigned to the hypervisor in the preferred embodiment.

Accordingly, each of the subchannel numbers A–E locates a non-shared SCB assigned to either the hypervisor (IID=0) or one of the OSs (IID other than zero). And each of the subchannel numbers F–Z is assigned to a sharing set of SSCBs for enabling its represented I/O device to be shared by the hypervisor and by each of the OSs having IIDs 1–N. The shared device will also share one or more channels when the same CHPIDs are specified in each of the SSCBs in the sharing set, as is done in the preferred embodiment herein.

Although each of the SSCBs in a sharing set may be considered to provide a "subchannel image" of each other, because they all apply to the same subchannel, these SSCBs only represent "complete subchannel images" of each other when they all have the same CHPID (channel) assignments.

A sharing set may contain "partial subchannel images" when some SSCBs in the sharing set do not have all of the CHPIDs specified in other SSCBs in the set. But a sharing set supports "shared channels" when two or more SSCBs in the sharing set have the same CHPID to enable different OSs to share the same channel when accessing the I/O device represented by the sharing set.

In the preferred embodiment herein, every SSCB in the same sharing set has the same CHPIDs, but different sharing sets have different sets of CHPIDS, although they may have some or all CHPIDs in common.

Structure of an SSCB

FIG. 12 shows the content of each SSCB and SCB shown in FIG. 11 (SSCBs and SCBs are identical in field structure and differ in whether or not they are used in sharing sets). Some of the fields in the illustrated SSCB/SCB content are identical to fields in the SCHIB (subchannel information block) found in the prior S/390 architecture. However, novel fields provided in the SSCB/SCB herein include an IID field, a chain pointer field, a QID field, an I/O interpretation control bit field (INCB), LCUCB number field, and an SSCB number field, which are defined as follows:

a. The INCB (I/O interpretation control bit) field indicates whether the subchannel image is enabled for instruction and interruption interpretation or not.

b. The chain pointer field in the SSCB allows it to be chained into any of several types of queues which perform a function used for selecting SSCBs, such as a "start subchannel work queue" and a "device interruption queue", such as the queues shown in FIG. 15.

c. The QID field content specifies a particular queue containing the SSCB (which applies to the pointer value in the chain pointer field).

d. The IID field contains the assigned IID value.

e. The LCUCB number field contains the logical control unit identifier of the LCUCB associated with the SSCB. The LCUCB number and IID fields are used in combination to locate the associated LCUCB in I/O subsystem storage.

f. The SSCB number field contains the related SSCB (or SCB) value.

The IID field and the SSCB number field contents are provided for verification checking. The values in these fields are implied by the location of the respective SSCB in the array shown in FIG. 11 (so theoretically these values are not needed to specify the SSCB or SCB).

The SSCB number field is required by this invention to contain the same SSCB number in every SSCB in the same sharing set.

The other fields in the SSCB/SCB shown in FIG. 12 may be the same as the fields defined in the prior art SCHIB in the ESA/390 Principles of Operation (previously cited herein), and they may also be provided in each SSCB. However, some of these fields found in the prior SCHIB are used in a novel manner by this invention, such as the following:

A valid field V indicates if the image represented by the SSCB is valid and may be used; if invalid, the represented image of the device cannot be accessed by the assigned OS (represented by its assigned IID). However, the same device may be accessed by other OSs having valid images of the same subchannel indicating by a valid state (having their valid bit=1, but assigned different IIDs). Hence, the V bit in each subchannel image can be set to either one or zero in order to allow only selected OSs to request I/O operations of the corresponding I/O device.

In the preferred embodiment each SSCB or SCB contains fields for up to eight CHPIDs (CHPID-0 to CHPID-7), which allows the device represented by a SCB or SSCB to be accessed through any of up to eight different channels represented by these CHPIDs. When a device is being selected for a communication operation (such as when a device is being started or is being reset), some of its CHPID-specified channels may not be usable by the requesting OS, due to being busy with other devices, or merely not being currently operational. When a channel is found available, it is assigned as the currently selected channel path to the device represented by the SCB or SSCB.

Enabled bit, E, indicates whether I/O operations may be performed by the image represented by this SSCB. The E bit value may be different for the different SSCB images in the same sharing set.

I/O Interruption subclass code, ISC, indicates the interrupt subclass used for an I/O interruption provided for the image represented by this SSCB. The ISC values may be different for the different SSCB images in the same sharing set.

Logical Path Mask, LPM, indicates the logical availability of the channels specified by the CHPIDs in the SSCB for accessing the I/O device specified by the subchannel number in the SSCB. The LPM field value may be different for the different SSCB images in the same sharing set.

Path Available Mask, PAM, has 8 bits which respectively indicate the physical availability of each of the installed channels specified in the CHPIDs 1–8 fields in the SSCB for use by the I/O device specified in the subchannel number field. The PAM field value may be different for the different SSCB images in the same sharing set.

A DB (device busy) field indicates whether the last request in the current logical channel path for this SSCB has encountered any device busy condition for which a device end condition has not yet been received. The DB field value may be different for the different SSCB images in the same sharing set.

An allegiance field, ALLEG, indicates for the currently assigned channel path for this SSCB image which, if any, of the following allegiance states apply: 0. no allegiance, 1. active allegiance, 2. dedicated allegiance, or 3. working allegiance. The ALLEG field value may be different for the different SSCB images in the same sharing set.

These and other subchannel control fields provide the I/O subsystem with the capability of independently keeping track of each subchannel image's state and attributes. For example, items such as path selection management, path availability, device busy conditions, and allegiances can all be handled independently for each subchannel image within each sharing set.

Generally, the IID values in the SSCBs (and in any SCBs) are established at the time a I/O subsystem is initialized, or is reconfigured. When a software hypervisor is used with the CEC, control blocks associated with unshared subchannels (SCBs) are set to an IID value equal to zero, which allows the software hypervisor to control the I/O operations performed on these subchannels. But when a microcoded hypervisor is used with the CEC, control blocks associated with unshared subchannels (SCBs) are set to an IID value of an OS when a channel is varied online to that OS.

Further, if the same channel (CHPID) is specified in all SSCBs in a sharing set, that channel is shared by all OSs in the CEC for making a connection to the device represented by the sharing set of SSCBs. Thus, the IID assignments to all SSCBs in a sharing set enable all OSs in the CEC to share any channel specified by a CHPID found in all the SSCBs in the sharing set.

Isolation of Shared Channels and Shared Subchannels

Each of the OSs can operate a physical channel or a physical I/O device independently of the other OSs through the use of images of these channels and subchannels. No software coordination is required among these OSs in their use of the physical channels and I/O devices. All coordination among the OSs is automatically done through the image control blocks such as the CHCBs and SSCBs, and other image control blocks to be described herein.

The identification of control blocks with different IIDs enables the different OSs to independently issue subchannel related I/O instructions to the same shared I/O device through shared channels without physical

interference among the OSs, and with the information transmitted for each each OS being completely secure from the operations done with the same physical transmission media by the other OSs. It must be understood that information residing on a shared I/O device is the responsibility for each OS to make secure from other OSs.

Although the different OSs view the same subchannel numbers, (same I/O device identifiers) and same CHPIDs (channel identifiers) in the sharing sets, no OS can access the channel or subchannel image associated with another OS, because locating any channel image or subchannel image requires an IID value not viewed by the OSs (no OS has access to the IID values), and can only be accessed by the hypervisor, CPUs, and I/O subsystem. Accordingly, the lack of access to the IID values, and the inability of the OSs to access the I/O subsystem control blocks is enforced by putting the IIDs in control blocks in storage media that are not addressable by any OS executed instruction, which prevents any operation by one OS from interfering with the operations of any other OS.

Address Generation for SSCBs and SCBs

The address of a required SSCB or SCB is obtained by the microcode executed for an OS instruction requesting an I/O operation. The SSCBs in FIG. 11 are in the storage of the I/O subsystem, which is not addressable by OS executed instructions (which in this embodiment are in the system area of an IBM S/390 compatible system). An SSCB is accessed by generating a storage address given a subchannel number and an IID value.

The address of a required SCB or SSCB can be determined by: multiplying its subchannel number by the size of each SCB or SSCB and adding this result to the proper base address. There is a single proper base address which is used for all SCBs, and there are multiple base addresses (one for each IID provided in the CEC) that are used for SSCBs. The proper base address for a SSCB is the one which corresponds to the IID field in the SSCB. In the preferred embodiment, all the base addresses are calculated at the time the I/O subsystem is initialized. This makes the calculation of a SCB or SSCB address much faster than if the base address were to be calculated each time it is needed. The calculation of the base addresses is dependent on whether a single, continuous range of storage addresses are used for all SCBs and SSCBs or whether multiple, continuous ranges of storage addresses are used for the range of SCBs and ranges and SSCBs associated with different IIDs.

Address Generation for CHCBs and LCUCBs can be done in a manner similar to SCBs/SSCBs. For CHCBs, it is a CHPID and IID which uniquely identify a CHCB. For LCUCBs, it is a LCUCB number and IID which uniquely identify a LCUCB.

Expansion of the Number of Channels and Subchannels

The invention can expand the effective number of available channel and subchannel images in a CEC far beyond the maximum number provided by the largest number available from the number of bits in the CHPID and subchannel number values. The maximum number of channel images available to a CEC is equal to the maximum number of channels multiplied by the number of IIDs in the CEC. And, the maximum number of subchannel images available to a CEC is equal to the maximum number of subchannels multiplied by the number of IIDs in the CEC. This occurs because each CHPID and each subchannel number is replicated for each IID value.

Use of Dynamic Switch

Referring to FIGS. 16A and 16B, a physical channel 65 may or may not be connected through a dynamic switch 62 to a physical CU 60 (where plural logical CUs may exist within the physical CU). If the channel is connected to switch 62, the physical channel path is considered a physical channel link 63 between the CEC I/O subsystem and switch 62; and then a physical control unit link 61 connected between switch 62 and the physical CU.

If switch 62 is not used, the physical channel path is considered the physical channel link between the CEC I/O subsystem and the physical CU (where plural logical CUs may exist within the physical CU).

FIG. 16B shows dynamic switch 62 connected to the physical channel links 63-0 through 63-P of physical channels 65-0 through 65-P. Control unit (CU) ports of switch 62 are connected to physical CU links 61-0 through 61-L which connect to ports of a physical CU box 60. Plural logical CUs exists within physical CU box 60, each of which is able to use all the ports of the physical CU box. CU box 60 connects to physical I/O devices A-E through Y-Z. (FIGS. 16A and 16B together show an integrated embodiment of the invention having different types of control blocks providing various types of IID-associated images representing OS-shared channels, devices and logical control units.)

Logical Control Unit Images (LCUCBs)

A physical control unit (physical CU) is an entity generally found in an electronic box, or on card(s) and-/or chip(s) within a box, to control the operation of one or more connected I/O devices, such as DASD, tape, printer, display, etc.

The ESCON I/O Interface architecture provides for a plurality of logical control units (logical CUs) to exist within a physical CU. A logical CU provides the functions and has the logical appearance of a control unit. When plural logical CUs do not exist within a physical CU, a single logical CU is said to exist in the physical CU. Logical CUs within a physical CU can be of the same general type (e.g. for control of DASD) or of different general types (e.g. one for control of DASD, another for control of printers, etc).

Each of the logical CUs within a physical CU may use all of the ports which exist for the physical CU. A logical CU is uniquely identified within a physical CU by the logical CU address which is included in the frame header sent from a channel to a logical control unit and in the frame header sent from a logical control unit to a channel.

The information and controls in the I/O subsystem related to a logical CU is kept in a LCUCB. The I/O subsystem identifies a LCUCB by the use of a logical CU identifier (LCUCB number).

This invention provides a set of logical CU images for each logical CU by means of a "sharing set" of LCUCBs. The LCUCBs in a sharing set are respectively assigned to different IIDs (representing different OSs). This enables up to all OSs in a CEC to share the same logical CU because information and controls related to the logical CU (such as information and controls related to logical paths between channel images

and the logical CU) are maintained separately for each image of the logical CU.

Each LCUCB is a sharing set has the same LCUCB number and is uniquely identified by the combination of a LCUCB number and an IID. FIG. 13 shows an example an array of LCUCBs, similar to the arrays of CHCBs and SCBs/SSCBs shown in FIGS. 8 and 11, respectively. The LCUCBs in the array are arranged by LCUCB number 0–K in the vertical direction, and by IID numbers 0–N in the horizontal direction.

FIG. 16A also shows sharing sets of of LCUCBs 67-0(0)–67-0(N) through 67-K(0)–67-K(N), in which all LCUCBs are in sharing sets (no unshared LCUCB is used). In FIG. 16A, each sharing set of LCUCBs, e.g. 67-0(0) through 67-0(N), is associated with one or more sharing sets of SSCBs. SSCB-A(0)–SSCB-A(N) is one such example of these sharing sets of SSCBs, and represents the same device 71A that is connected to the associated logical CU shown in FIG. 16B.

Structure of an LCUCB

FIG. 14 shows the content of each LCUCB shown in FIG. 13. The first row in the LCUCB contains a LCUCB number field for the number assigned to this LCUCB, an IID field for the IID assigned to the respective LCUCB, and a logical CU address field which identifies the logical CU within the physical CU. Each LCUCB is the header of a busy queue comprising SSCBs (and/or SCBs) currently enqueued on the busy queue and is used in locating the top and bottom elements in the queue. Thus, each LCUCB controls a queue of subchannel images currently function pending and delayed because of busy conditions.

The "V" field indicates if LCUCB is valid. V=1 indicates the LCUCB is valid. V=0 indicates that the LCUCB is not valid.

The "IID" field contains the IID assigned to the associated CU image.

The "logical CU address" field contains the logical CU address which identifies the logical CU within the physical CU.

The "LCUCB number" is the logical CU identifier for the I/O subsystem.

The "CU busy queue count field" contains the current length of the busy queue. The length of the queue is determined by the number of subchannel images on the associated busy queue that are function pending and delayed because of busy conditions.

Top and bottom pointer fields are for containing addresses of the top and bottom queue elements in the CU queue.

The "summation of queue counts" field adds the queue-count field to the summation at the time a subchannel image is added to the specified busy queue.

The "summation of enqueues" field contains an unsigned binary count of the number of times a subchannel image is added to the specified busy queue.

CHPIDs 0–7 are the same as the up to eight CHPIDs in each of the SSCBs for devices associated with the logical CU represented by the respective LCUCB.

Following the above defined fields within the LCUCB are eight subset fields, of which each subset is associated with a different one of the eight CHPID fields. Each subset contains the following fields:

Field B contains a busy indication for the associated logical CU image.

Field E indicates if a request exists for establishment of a logical path between the associated logical CU image and channel image.

Field R indicates if a request exists for the removal of a logical path between the associated logical CU image and channel image.

Field S indicates if a request exists for a device-level-system-reset between the associated logical CU image and channel image.

Field L indicates if a currently established logical path exists between the associated logical CU image and channel image.

The "physical channel link address" field, and the "physical CU link address" field can have their content combined with the contents of the preceding IID and logical CU address field to provide the identity of the logical path which could exist between the associated channel image and logical CU image.

A "switch busy count" field contains a count of the number of times an initial selection sequence for a start or halt function resulted in a switch-busy response on the corresponding logical channel path. Each switch-busy-count field corresponds one-for-one, by relative position, with the PIM bits of the subchannels associated with the logical CU.

The "CU busy count" field contains a count of the number of times an initial selection sequence for a start or halt function resulted in a control-unit-busy response on the corresponding logical channel path. Each control-unit-busy-count field corresponds one-for-one, by relative position, with the PIM bits of the subchannels associated with the logical CU.

The "success count" field contains a count of the number of times an initial selection sequence for a start function resulted in the device accepting the first command of the channel program on the corresponding logical channel path. Each success-count field corresponds one-for-one, by relative position, with the PIM bits of the subchannels associated with the logical CU.

Control Unit Logical Path Control Block (CULPCB)

A logical path must be established between a channel and a logical CU before communication to an I/O device attached to that logical CU can take place. This invention expands the identification of a logical path (LP) to include the IID corresponding to a channel image. Thus, a unique LP needs to be established between each image of a channel and a logical CU before communication to an I/O device attached to that logical CU can take place using each of the respective channel images. A single LP is uniquely identified by the combination of the physical channel link address, physical CU link address, IID, and logical CU address, with respect to either a physical channel or a control unit port.

The information and controls in an I/O control unit (CU) related to an established LP are kept in a "Control Unit Logical Path Control Block" (CULPCB). The number of CULPCBs which exist in the I/O control unit's storage determines the maximum number of LPs that a CU can have established at any one time. The maximum number of CULPCBs which exist in the I/O control unit's storage is open-ended since it may be a variable number.

When a channel image requests that a LP be established between the channel image and a logical CU (using the establish-logical-path procedure), the CU attempts to locate an available CULPCB which can be

associated with the specified LP. A CULPCB currently associated with any other established LP is not considered available by the CU. If an available CULPCB can be located, the CU may respond to the channel image that the LP has been established. If an available CULPCB can not be located, the CU responds to the channel image that the LP has not been established. Once a LP is established, the CULPCB associated with this established LP is no longer available. The CULPCB may later become available if the established LP which the CULPCB is associated with is later removed.

A CULPCB associated with an established LP within a CU is identified by the same identifiers that identify a LP with respect to a control unit port. That is, a CULPCB is uniquely identified by the combination of a physical channel link address, physical CU link address, IID, logical CU address, and a control unit port identifier (CU port number). Once a CULPCB becomes associated with an established LP, that CULPCB becomes associated with the channel image, logical CU, and control unit port corresponding to the established LP.

An available CULPCB within a physical CU may be conditionally available for the establishment of a LP depending upon the identity of the logical path and/or control unit port. For example, a CULPCB may be restricted for association with a LP which is identified by a subset of logical CU address values which are valid within a physical CU.

FIG. 16B show an example of a physical CU 60 (packaged in a single box) having a plurality of logical CUs 60-0 through 60-K. Each logical CU may have a different plurality of CULPCBs associated with the logical CU. For example, logical CU 60-0 has associated with it CULPCBs 60-0(1) through 60-0(G) and logical CU 60-K has associated with it CULPCBs 60-K(1) through 60-K(H).

Structure of an LCUCB

FIG. 4 illustrates an example of the structure of a control unit logical path control block (CULPCB), which is provided in the hardware/microcode of a physical CU to represent a single LP which is established between a channel image and a logical CU.

In FIG. 4, the first row in the CULPCB contains all of the component identifiers for the CULPCB (which are also the component identifiers of the associated established LP and logical channel path). Each of the other rows in the CULPCB represents information about an I/O device connected to the associated logical CU, for example of I/O devices 71A through 71E connected to logical CU 0 (logical CU address =0) in FIG. 16B. The I/O information includes allegiance indicators for the I/O device as defined in the S/390 Principles of Operation (previously cited herein), a PGID (assigned for the I/O device by an OS), and model-dependent control fields tailored to the particular logical CU and device.

The PGID (multiple path group identifier) references a location in the CU storage that defines a set of logical channel paths selectable by the CU, given LPs are established, when responding to a requesting OS on behalf of an I/O device. The channel path group includes logical channel paths connecting the same operating system to the plurality of ports of a CU, and the group is assigned the PGID by an OS. Because this invention provides each OS with separate and unique logical channel paths to access a shared I/O device using shared channels, each OS may group together logical channel paths to a device using its own desired path-group identifier (PGID).

Image Reset

Prior to this invention, a logical resource partition reset command issued by the hypervisor to the I/O subsystem caused a reset of all controls for a logical resource partition along with a reset of all control units and devices connected via the logical paths associated with the logical resource partition. The command included a list of I/O subsystem resources dedicated to the logical resource partition rather than directly specifying the identity of the logical resource partition. The control units and devices were reset by issuing device-level-system-reset commands over only those established logical paths that were associated with the channels dedicated to the specified logical resource partition.

A logical resource partition reset command was issued, for example, when activating, inactivating, performing a system reset, or performing an initial program load (IPL) for the logical resource partition.

With this invention, an "image reset" command issued by the hypervisor to the I/O subsystem causes a reset of all controls for a specified IID along with a reset of all control units and devices connected via the logical paths associated with the specified IID. The control units and devices are reset, as for example when a system reset of a logical resource partition is performed, by issuing device-level-system-reset commands over only those established logical paths that are associated with the specified IID.

Novel to this invention is that the image reset command reinitializes only those controls for shared I/O resources within the I/O subsystem and control units which are associated with the specified IID. That is, only controls within SSCBs, LCUCBs, and CHCBs which are associated with the specified IID are reinitialized. SSCBs, LCUCBs, and CHCBs associated with other IIDs are not affected. Additionally, only controls within CULPCBs associated with established logical paths over which a device-level-system-reset command is received are reinitialized. Other CULPCBs are not affected.

When the hypervisor issues the image reset command, an indication is also included which specifies whether the target IID is to be placed in the activated state or inactivated state at the completion of the image reset command. The hypervisor specifies that the target IID be placed in the activated state, for example, when activating, performing a system reset, or performing an initial program load (IPL) for a logical resource partition. The hypervisor specifies that the target IID be placed in the inactivated state when inactivating a logical resource partition.

An activated IID is available for use by an OS. An inactivated IID is not available for use by any OS, although it may later be activated for use by an OS.

The configuration control block (CCB) shown in FIG. 9 is used by the I/O subsystem to control the current activated/inactivated state of each IID. When an image reset command is issued to the I/O subsystem, the bit corresponding to the target IID is set to one when the IID is to be placed in the activated state or is set to zero when the IID is to be placed in the inactivated state.

An activate/inactivate indication was not included with the logical resource partition reset command used prior to this invention. Instead, the I/O subsystem assumed that all logical resource partitions were always activated.

Novel to this invention is that the activate/inactivate indication included with the image reset command enables the I/O subsystem to remove all established logical paths associated with an inactivated IID because the I/O subsystem is now given the knowledge of which IIDs are activated and which IIDs are inactivated. It is important to remove established logical paths for an inactivated IID so that CULPCBs within the storage of I/O control units can be made available for the establishment of other logical paths.

When a previously inactivated IID is activated via the image reset command, the I/O subsystem attempts to establish all logical paths associated with the target IID. When a previously activated IID is inactivated via the image reset command, the I/O subsystem removes all logical paths associated with the target IID. When a previously activated IID is activated via the image reset command, the I/O subsystem sends device-level-system-reset commands over all established logical paths associated with the target IID in order to reinitialized controls within control unit's CULPCBs associated with the established logical paths.

The image reset command is part of both the Service Call Logical Processor (SCLP) instruction and Processor Controller CAll (PCCALL) instruction which passes a control block containing the target IID and activate/inactivate indication. This command is issued only by the hypervisor.

I/O Instructions

Most of the I/O instructions to the I/O subsystem use the SSCBs, CHCBs, LCUCBs, and CULPCBs.

Start Subchannel Instruction Example

An example of an I/O instruction is shown in FIGS. 17 and 18, which provide a flow diagram of the execution of the S/390 "start subchannel" (SSCH) instruction. The SSCH instruction is issued by any operating system (OS) in the CEC in FIG. 1 executing in any resource partition and assigned an IID. The steps in the SSCH instruction execution are as follows:

101) Before an OS can use the invention on a CEC, a state description (SD) for the OS (shown in FIG. 6) is loaded into the memory of the CEC by the CEC hypervisor. The content of the SD defines resources for partition J and is assigned an IID value.

102) Step 102 loads the OS into the CEC memory assigned to resource partition J.

103) The OS is dispatched on a CPU in the CEC by the hypervisor.

104) The OS dispatches an application program as a task on the dispatched CPU.

105) The task requests an I/O operation of the OS by the task issuing an supervisor call (SVC) instruction, using application-to-OS program interface facilities such as a read, get, write, or put request.

106) The OS issues an SSCH instruction to start the I/O device to perform a requested operation.

107) CPU microcode responds to the SSCH instruction operation code by accessing the IID from the SD of the issuing OS and by obtaining the requested subchannel number from general register GR1. Then the microcode uses the IID and subchannel number to select a required SSCB in FIG. 12. This SSCB is the subchannel image used for the OS to access the desired I/O device.

108) The microcode tests the validity (V) bit and I/O interpretation control bit (INCB) in the SSCB to determine if it is a valid SSCB and whether it is operating in passthru mode, respectively. If the SSCB is valid and operating in passthru mode, the yes exit is taken to step 115. If the SSCB is not valid or is not operating in passthru mode, the no exit is taken to step 109.

109) Using the subchannel number from general register GR1 and an IID=0 (i.e. the hypervisor's IID), a second attempt is made to select a SSCB (in FIG. 12) which represents the desired I/O device. The microcode tests the validity (V) bit and I/O interpretation control bit (INCB) in the SSCB to determine if it is a valid SSCB and whether it is operating in passthru mode, respectively. If the SSCB is valid and operating in passthru mode, the yes exit is taken to step 115 (i.e. the hypervisor has setup the SSCB assigned to itself such that it can be used by an OS in passthru mode). If the SSCB is not valid or is not operating in passthru mode, the no exit is taken to step 113.

113) The hypervisor intercepts the execution of the SIE instruction for OS-J. The hypervisor may either terminate the I/O instruction with an unsuccessful condition code (CC) (e.g. when the selected SSCB is invalid) or may simulate the execution of the I/O instruction for OS-J (e.g. when the selected SSCB is not operating in passthru mode).

115) The CPU microcode accesses SSCB for OS-J to execute the synchronous portion of the SSCH instruction.

116) This step tests the SSCH instruction's condition code to determine if the CPU part of the SSCH instruction execution executed successfully, which involves the CPU microcode selecting an I/O processor (IOP). Using the information contained in the SSCB, the CPU performs the normal checks to determine which condition code (CC) to give to the SSCH instruction. If an unsuccessful CC is determined, step 117 is entered. If successful, step 119 is entered. 117) If the SSCH instruction's CC indicates the CPU did not execute it successfully, an exception is indicated which causes step 118 to be entered. 118) The hypervisor intercepts to determine why the CPU did not execute the SSCH instruction successfully, and takes action accordingly.

119) Then, the CPU enqueues the SSCB on the selected IOP's work queue contained in the I/O subsystem storage, and ends the CPU portion of the instruction. When the CPU executed the synchronous portion of the SSCH instruction successfully, an asynchronous portion of its execution begins with IOP operations on the work queue.

120) The work queue operates FIFO. An IOP dequeues the SSCB from the work queue when the SSCB rises to the top of the queue. Then, the IOP performs path selection by selecting a channel processor represented by one of the CHPIDs in the SSCB being used.

121) Once a channel processor is selected, the IOP issues a command to the channel processor to begin its I/O operations. The IOP command contains the IID of the SSCB, the CHPID, and the subchannel

number in the SSCB. Then, step 131 in FIG. 18 is entered. 131) Channel processor receives the IOP command and calculates the address of the SSCB using the received subchannel number and IID.

122) Channel processor constructs an ESCON command frame to send to the director (when one exists) and to the control unit. FIG. 5 illustrates the frame header, which includes the address information. The header includes the fields shown in FIG. 5. The LP identifiers are obtained from a LCUCB. The LCUCB associated with the SSCB is located using the LCUCB number and IID in the SSCB.

123) The channel processor transmits the frame through the director (when one exists) to the addressed logical control unit.

124) The CU receives the frame, examines it, and accesses the I/O device addressed in the frame. The logical CU maintains its controls for the I/O operation in the CULPCB which is located using the identifiers in the frame header and the control unit port identifier over which the frame was received. (The combination of the source link address and source logical address identify the channel image which sent the frame. The combination of the destination link address and destination logical address identify the logical control unit which is the target of the frame. The device address identifies the particular I/O device on the logical control unit.) Any requested data is accessed by the addressed I/O device. The CU then sends a command acceptance frame to the channel, using the frame-sending procedure described below. In the preferred embodiment, the ESCON protocol is used to transfer the data associated with the command frame. If a write request, data transmitted in subsequent frames is written by the device. If a read request, data read by the device is sent back to the CU for placement in return frames the CU will send to the channel processor.

126) In order to send a frame to the channel processor, the CU constructs a returning ESCON frame to send to the channel, which has a frame header similar to that in FIG. 5, except the content of the destination and source parts of the frame header are reversed. (The source link address in the frame is set to the physical CU link address. The source logical address is set to the logical CU address. The destination link address is set to the physical channel link address. The destination logical address is set to the IID value. The frame also contains the I/O device's corresponding device address.)

127) The channel processor receives the frame and examines it. The combination of the destination link address and destination logical address identify the OS-J channel image which is the target of the frame. The combination of the source link address and source logical address identify the logical control unit which sent the frame. The device address identifies the particular I/O device on the logical control unit.

128) Using the destination logical address (J[IID), source link address (physical CU link address), source logical address (logical CU address), and device address from the frame, the channel processor computes the address of the SSCB using a reverse lookup control block (RLCB). If the frame is a data frame, the data are stored in the program buffer. If the frame is a status frame, the channel processor places ending status in that SSCB.

129) After the status frame is received, the channel processor signals the IOP that the I/O operation is completed for this command, indicating the subchannel number and IID of the SSCB.

130) The IOP calculates the address of the SSCB using the subchannel number and the IID and places the SSCB on the bottom of an interruption queue for the interruption subclass indicated in a field of the SSCB. The interruption queue is contained in the I/O subsystem. This ends the IOP operations for the SSCH instruction. The interruption queue uses a FIFO structure. An interruption signal is sent to all CPUs in the CEC that the interruption is pending in an interruption queue.

131) The first CPU which is enabled for the I/O interrupt dequeues the SSCB from the interruption queue and constructs an interruption response block (IRB) in the system storage for OS-J when the I/O interruption takes place. The IRB contains the subchannel number but does not contain the IID.

Reset Channel Path Instruction

Another example novel to this invention is the "reset channel path" (RCHP) instruction, which prior to this invention reset all controls for the specified channel along with all control units and devices connected via the logical paths associated with the specified channel. With this invention, this instruction requires a specified IID and resets only those controls and logical paths associated with the channel image identified by the IID and CHPID.

When an OS issues the "reset channel path" instruction, the target channel (CHPID) is specified in GPR 1. This instruction is not executed interpretively via the SIE instruction, but is intercepted by the hypervisor which in turn provides the IID assigned to the channel image in GPR 1 in addition to the CHPID before issuing the RCHP instruction to the I/O subsystem. FIG. 19 shows the format of GPR1 provided by this invention. The, combination of the IID and CHPID values specify the channel image as the target of the reset function.

The I/O subsystem resets controls for the specified channel image and issues device-level-system-reset commands only for those established logical paths that are associated with the specified channel image by building frame headers that include the specified IID. All other channel images and all other established logical paths are not affected. Further, the I/O subsystem resets the controls (busy indications and allegiances) in only those subchannel images (SSCBs) associated with the specified channel image. Controls in subchannel images (SSCBs) associated with any other channel images are not affected.

Channel Reports

Certain channel reports are presented to an OS to report information about either a channel or a subchannel. A channel report consist of one or more channel report words (CRWs) chained together. Prior to this invention, these channel reports were presented to a single OS since the channels and subchannels were not shared. With this invention a mechanism is provided to present these channel reports either to a single OS or to multiple OSs when sharing subchannels and channels.

In some cases the channel report applies only to one of the OSs sharing the channel or subchannel. In other cases, the channel report must be presented to all OSs sharing the channel or subchannel.

FIG. 20 shows the format of the channel report word as provided by this invention.

When the channel report applies to all OSs, the I/O subsystem presents the report to the hypervisor in the same way as was done prior to this invention. The Image (I) bit in the channel report word is set to zero. Novel to this invention, the hypervisor in turn presents this report to all of the OSs sharing the channel or subchannel. An example of this type of report is a permanent failure in the channel hardware.

When the channel report applies to only one OS, this invention provides a means for the I/O subsystem to pass the IID assigned to the image for which the report is intended along with the report. The Image (I) bit is set to one. Further the Chaining (C) bit is also set to one and an additional channel report word is chained to the original which provides the IID value in the Reporting-Source ID field. The hypervisor in turn presents this report only to the OS associated with the IID without the chained CRW containing the IID and without the I bit being set to one. An example of this report could be the result of a RCHP instruction issued by the same OS.

Channel Reconfiguration

Prior to this invention, a channel configuration command issued by an OS or by manual means caused the system to physically vary the target channel offline or online. A channel that is varied online to an OS is available for use by that OS. A channel that is varied offline is not available to any OS.

With this invention, a channel configuration command issued by an OS causes the I/O channel subsystem to vary offline/online only the channel image associated with the OS. The channel configuration commands are part of the Service Call Logical Processor (SCLP) instruction which passes a control block containing the target channel. This instruction is not executed interpretively via the SIE instruction, but is intercepted by the hypervisor. The hypervisor in turn provides the IID assigned to the image in the control block in addition to the CHPID before issuing the channel configuration command to the I/O subsystem. The combination of the IID and CHPID values specify the channel image as the target of the configure command.

The I/O subsystem resets controls for the specified channel image and either removes logical paths (for vary offline) or establishes logical paths (for vary online) only for those logical paths that are associated with the specified channel image by building frame headers that include the specified IID. All other channel images and all other established logical paths are not affected. Further, the I/O subsystem resets the controls (busy indications and allegiances) and either turns off the appropriate path availability bit (for vary offline) or turns on the appropriate path availability bit (for vary online) in only those subchannel images (SSCBs) associated with the specified channel image. Controls in subchannel images (SSCBs) associated with any other channel images are not affected.

Many variations and modifications are shown which do not depart from the scope and spirit of the invention and will now become apparent to those of skill in the art. Thus, it should be understood that the above described embodiments have been provided by way of example rather than as a limitation.

What is claimed is:

1. A method for sharing input/output (I/O) resources among a plurality of programs in a computer electronic complex, each of said plurality of programs assigned an image identifier, said method including the steps of:

storing in said computer electronic complex a sharing set of input/output control blocks, each input/output control block of said sharing set of input/output blocks including an input/output resource identifier and an image identifier;

accessing said sharing set of input/output control blocks in said computer electronic complex in response to an input/output request of one of said plurality of programs;

accessing a control block in said sharing set of input-/output control blocks, accessed in the just previous step, with an image identifier of said one of said plurality of programs; and

storing states of input/output resources in said control block accessed in the just previous accessing step, whereby said control block presents separate images of the input/output resources to each of said plurality of programs.

2. A method of sharing I/O resources by a plurality of programs as defined in claim 1, wherein said computer electronic complex includes a single operating system program with multiple categories of programs in which each category has one or more image identifiers assigned to it and including the steps of associating an image identifier with each executing program in each category and utilizing an identifier of the I/O resource to select a set of input control blocks and utilizing an image identifier associated with the program to select a required input control block in the required set to enable independent use of the I/O resource by programs in the different categories using different input control blocks in the set of the input control blocks.

3. A method of sharing I/O resources by a plurality of programs as defined in claim 1, wherein said computer electronic complex includes multiple operating systems in which each operating system has one category and each category has one or more image identifiers assigned to it and including the further step of utilizing an identifier of the I/O resource to select a required set of input control blocks and utilizing an image identifier associated with the program to select a required input control block in the requested set to enable independent use of the I/O resource by programs in the different operating systems using different input control blocks in the set of input control blocks.

4. A method of sharing I/O resources by a plurality of programs as defined in claim 1, comprising the steps of:

accessing the image identifiers and control blocks by means of internal code which is not accessible to the executing programs.

5. A method of sharing I/O resources by a plurality of programs as defined in claim 1, comprising the steps of:

accessing the image identifiers by means of system software used by a computer electronic complex hypervisor, and accessing the control blocks by means of internal code, in which the image identifiers and control blocks are not accessible to any programs executing in the computer electronic complex.

6. A method of sharing I/O resources by a plurality of programs as defined in claim 1, comprising the steps of:

providing plural categories of programs in which a hypervisor is one of the categories.

7. A method of sharing I/O resources by a plurality of programs as defined in claim 1, comprising the steps of:

providing plural categories of programs in which operating systems (OSs) are one or more of the categories.

8. A method of sharing I/O resources by a plurality of programs as defined in claim 1, comprising the steps of:

communicating an I/O request directly from one of the programs to an I/O subsystem using at least one shared I/O resource without hypervisor intervention;

performing by the I/O subsystem of the I/O request using the shared I/O resource without hypervisor intervention; and

responding to the requesting program without hypervisor intervention.

9. A method of sharing I/O resources by a plurality of operating systems (OSs), comprising the steps of:

storing an image identifier (IID) in a resource identifying control block (SD) for each of plural OSs operating in a computer electronic complex (CEC);

structuring within an I/O-control storage for the CEC of a set of I/O control blocks (CBs) for each of plural I/O resources of the CEC, and associating each CB in the set with a different IID; and

accessing a required CB by an I/O processor for an I/O operation by selecting a set of CBs with an identifier of an I/O resource and by selecting a required CB in a selected set with the IID stored in the SD of an OS requesting the I/O operation, and sharing the I/O resource among the OSs associated with the IIDs of the different CBs in the set.

10. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 9, further comprising the steps of:

assigning one of the IID values with a hypervisor of the CEC (instead of with any OS) to enable the hypervisor to have its respective image of the I/O resource to permit the hypervisor to use the I/O resource independently of any OS and concurrently with the OSs.

11. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 9, further comprising the steps of:

structuring within an I/O-control storage for the CEC of a single I/O control block (CB) for any I/O resource and associating the CB with a hypervisor IID or an OS IID to provide a single image for an unshared I/O resource only to the hypervisor or OS, respectively, for enabling the hypervisor or OS, respectively, to control the unshared I/O resource in order to intermix CEC I/O operations for shared resources and for unshared resources, for which the OSs directly use shared resources without hypervisor intervention, and for which the hypervisor or the OSs can directly use unshared resources.

12. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 9, further comprising the steps of:

setting states independently in different control blocks for the concurrently executing programs of different programs to enable independent control for the different programs over the I/O resource being shared by the OSs.

13. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 12, further comprising the steps of:

communicating a representation of the IID of each I/O operation from the CEC to an I/O control unit (as the I/O resource for the I/O operation), and storing the representation of the IID by the I/O control unit to enable the I/O control unit to respond to the OS for the I/O request.

14. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 13, further comprising the steps of:

responding by the I/O control unit to the CEC for the I/O operation by the I/O control unit accessing the representation of the IID stored for the I/O operation and signalling the representation of the IID to an I/O subsystem in the CEC;

selecting by the I/O subsystem of a required set of CBs for the I/O resource, and selecting a required CB in the set for the OS requesting the I/O operation with the I/O control unit; and

queuing the required CB into an interruption queue for the OS requesting the I/O operation to enable a CPU executing the OS to handle the I/O interruption without any access to the IID to isolate the OSs from the IIDs and provide security of IIDs from the OSs.

15. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 9, further comprising the step of:

resetting the state of an image of an I/O resource for an OS (represented by a CB of the I/O resource associated with a requested IID in the CEC) without affecting the state of any other CB for the same I/O resource or the state of any CB associated with any other I/O resource by accessing and setting to an initial state only the CB of the IID and resource identified to be reset.

16. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 9, further comprising the step of:

resetting the state of CBs for all I/O resources (represented by all CBs of the I/O resources associated with a requested IID in the CEC) without affecting the state of I/O resources associated with any other IID by searching through the CBs of the CEC and setting to an initial state the CBs for all I/O resources found for the requested IID.

17. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 9, further comprising the step of:

structuring within an I/O-control storage for the CEC of a sharing set of I/O control blocks (CHCBs) for each of a plurality of physical I/O channels (each physical I/O channel being the I/O resource for a set of CHCBs) and associating each CHCB in the set with a different IID to provide a different image of the physical I/O channel to each of the OSs for enabling independent control over each physical I/O channel by the OSs sharing the physical channel.

18. A method of sharing physical I/O channels and I/O devices among a plurality of operating systems

(OSs) as defined in claim 17, further comprising the steps of:

identifying in each CHCB of an offline/online field, and

setting the offline/online field independently in each of the CHCBs in the sharing set to enable each of the OSs to independently control whether an associated image of the physical channel (provided by an associated CHCB) is online or offline.

19. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 17, further comprising the step of:

specifying a channel path identifier (CHPID) as an identifier of each physical channel, and the physical channel being the I/O resource for a sharing set of CBs.

20. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 17, further comprising the step of:

structuring within an I/O-control storage for the CEC of a single I/O control block (CB) for any physical I/O channel (the channel being the I/O resource for the single CB) and associating the CB with a hypervisor IID or an OS IID to provide a single image for an unshared physical channel only to the hypervisor or the OS, respectively, for enabling only the hypervisor or the OS, respectively, to control each unshared physical channel in order to intermix shared channels and unshared channels in a CEC for enabling any of the plurality of OSs to use shared channels without hypervisor intervention, and for which the hypervisor or the OSs can directly use unshared channels.

21. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 9, further comprising the steps of:

structuring within an I/O-control storage for the CEC of a set of I/O control blocks (CBs) for each of a plurality of subchannels in which each subchannel represents an I/O device (the subchannel being the I/O resource for the set of CBs), using an identifier of the subchannel as an identifier of the set of CBs, and associating each CB in the set with a different IID to provide a different image of each subchannel to each of the OSs for enabling independent control over each I/O device by each of the OSs.

22. A method of sharing I/O resources by a plurality of operating systems (OSs) as defined in claim 9, further comprising the steps of:

structuring within an I/O-control storage for the CEC of a set of I/O control blocks (CBs) for each image of a I/O control unit (CU) connectable to the CEC (as one of the I/O resources) and associating each CB in the set with a different IID to provide a different image to the respective OSs of the CU for enabling independent control over the CU by each of the OSs.

23. A method of efficiently sharing I/O channels, I/O control units, and I/O devices by a plurality of operating systems (OSs), comprising the steps of:

storing in processor storage one or more special control blocks (SDs) containing OS identifiers (IIDs) for use in I/O resource sharing operations of a computer electronic complex (CEC);

storing in I/O storage a sharing set of subchannel control blocks (SSCBs) for each subchannel (for a

shared I/O device), each SSCB in the set having a different IID;

storing in the I/O storage of a sharing set of channel control blocks (CHCBs) for the same I/O channel (shared channel) in which each CHCB in the set has a different IID;

storing in the I/O storage a sharing set of logical control unit control blocks (LCUCBs) for each logical I/O control unit in which each LCUCB in the set has a different IID; and

controlling an I/O operation for a requesting OS by obtaining an IID for the OS and selecting an SSCB with the IID in a sharing set for a required subchannel I/O resource, and selecting a LCUCB with the IID in a sharing set for a required control unit I/O resource, and selecting a CHCB with the IID in a sharing set for a required channel I/O resource, thereby enabling the OS to share the required I/O channel and the required I/O control unit and the required I/O device with at least one other OS operating in the CEC.

24. A method of sharing I/O channels and I/O devices among a plurality of operating systems (OSs) as defined in claim 23, further comprising the steps of:

generating by the CEC for the requesting OS of a frame header containing a specified logical path from the CEC through the required channel to a control unit associated with a device specified in the required subchannel, the logical path comprising: a representation of the IID of the requesting OS, an address for the required channel, an address associated with a required control unit (CU);

transmitting the generated frame header to the CU using the available logical path; and

storing in a CU storage of the logical path for use by the CU in later responding to the requesting OS regarding the I/O operation.

25. A method of sharing I/O channels and I/O devices among a plurality of operating systems (OSs) as defined in claim 24, further comprising the steps of:

also storing in the CU storage of a group of addresses for logical paths associated with each of plural IIDs for the OSs in a CEC making I/O requests to the CU, and assigning a path group identifier (PGID) in the CU storage for each group of logical paths for each OS in the CEC with which the CU can communicate;

recognizing by the CU of a need to provide a response to an OS for an I/O request made to the CU by the OS, and accessing by the CU of the PGID for the OS to access the group of logical paths for the OS in the CU storage;

selecting in the CU storage of an available logical path in the accessed group of logical paths for the accessed PGID;

generating by the CU of a responding frame header containing the available logical path for the accessed PGID containing a representation of an IID of the OS to receive the response;

transmitting the responding frame header to the CEC using the available logical path; and

selecting in the receiving CEC of a sharing set of SSCBs for the subchannel of the device having an address in the responding frame, and selecting of an SSCB in the sharing set with the representation of the IID in the logical path in the responding frame header.

IBMPSI01805102

26. A method of sharing I/O channels and I/O devices among a plurality of operating systems (OSs) as defined in claim 25, further comprising the steps of:

identifying in each SSCB of an interruption queue to be used by the SSCB of multiple subclass interruption queues provided in the CEC; and

choosing for the queuing step of the subclass interruption queue identified in the selected SSCB.

27. A method of sharing I/O resources among a plurality of operating systems (OSs) as defined in claim 9, further comprising the steps of:

communicating by an I/O subsystem to a hypervisor of an occurrence of a special condition in the I/O subsystem, the communication including an IID and an identification of each resource affected by the special condition and an indication that a report about the special condition is to be forwarded to the OS assigned the IID; and

forwarding by the hypervisor of the report containing information about the special condition to the OS.

28. A method of sharing I/O resources among a plurality of operating-systems (OSs) as defined in claim 9, further comprising the steps of:

communicating by mean of an I/O subsystem to a hypervisor an occurrence of a special condition in the I/O subsystem, the communication including an identification of each resource affected by the special condition and an indication that a report about the special condition is to be forwarded to each OS sharing the I/O resource; and

forwarding by the hypervisor of the report containing information about the special condition to each OS sharing the I/O resource.

29. A method of sharing I/O resources among a plurality of operating systems (OSs) as defined in claim 9, further comprising the steps of:

activating a logical resource partition (LPAR) in the CEC for use by an OS with an image-reset command;

communicating to the I/O subsystem that the LPAR is to be activated; and

establishing representations of logical paths (LPs) in the I/O subsystem and affected I/O control units for the LPAR being activated.

30. A method of sharing I/O resources among a plurality of operating systems (OSs) as defined in claim 9, further comprising the steps of:

inactivating a logical resource partition (LPAR) in the CEC with an image-reset command;

communicating to the I/O subsystem the IID of the LPAR with an indication the LPAR is being inactivated; and

removing representations of logical paths (LPs) from the I/O subsystem and from affected I/O control units for the LPAR being inactivated.

* * * * *

EXHIBIT 23

MORE THAN ONE MILLION COPIES SOLD

BUILT TO LAST

SUCCESSFUL HABITS OF VISIONARY COMPANIES

JIM COLLINS
(Bestselling author of GOOD TO GREAT)
JERRY I. PORRAS

To Joanne and Charlene

A hardcover edition of this book was published in 1994 by HarperBusiness, an imprint of HarperCollins Publishers.

HarperCollins books may be purchased for educational, business, or sales promotional use. For information please write: Special Markets Department, HarperCollins Publishers Inc., 10 East 53rd Street, New York, NY 10022.

Designed by Alison Lew
Illustrations by Irving Perkins Associates, Inc.

The Library of Congress has catalogued the hardcover edition as follows:
Collins, James C. (James Charles)
Built to last : successful habits of visionary companies / James C. Collins and Jerry I. Porras.
p. cm.
Includes index.
ISBN 0-88730-671-3
1. Success in business—United States. 2. Industrial management—United States. 3. Entrepreneurship—United States. I. Porras, Jerry I.
II. Title
HF5386.C735 1994
658—dc20 94-20571

ISBN-10: 0-06-051640-2 (pbk.)
ISBN-13: 978-0-06-051640-6 (pbk.)
07 08 09 / RRD / 29 28 27 26 25 24

We see a similar pattern at Walt Disney Company, which has stimulated progress throughout its history by making bold—and often risky—commitments to audacious projects. In 1934, Walt Disney aimed to do something never before done in the movie industry: create a successful full-length animated feature film. In creating *Snow White*, Disney invested most of the company's resources and defied those in the industry that called it "Disney's Folly." After all, who would want to see a full-length feature cartoon? Two decades later, after a string of full-length animated films, including *Pinocchio*, *Fantasia*, and *Bambi*, Disney made yet another risky commitment to one of "Walt's screwy ideas": to build a radically new kind of amusement park, later to become known to all of us as Disneyland. In the 1960s, Disney repeated the process again, with a commitment to fulfill Walt's dying dream: EPCOT center in Florida.[36] Walt's brother, Roy, carried the commitment through, according to Michael Eisner:

> He virtually gave his life to fulfill his brother's dream of building Walt Disney World. He gave up his much deserved retirement, infused the park throughout with Disney quality, and saw the project through to completion, personally cutting the ribbon on opening day. He died within two months of that momentous event.[37]

Columbia Pictures, in contrast, did very little that was bold, visionary, or risky. It produced B-grade movies during the 1930s and 1940s. During the 1950s and 1960s, it made some good films, but was apparently unwilling to make committing moves into the future. While Disney was pushing forward into EPCOT, Columbia was being run by people who saw themselves "first, last, and always . . . as investors, not managers."[38] And whereas Columbia was eventually acquired in the early 1980s, Disney came roaring back after it defeated a set of hostile raiders and pursued new bold adventures, such as Japan Disney and EuroDisney.

IBM, like Disney, pulled ahead of rival Burroughs at critical junctures in its history via the mechanism of tangible—and, at times, risky—commitments to audacious goals. In particular, we point to IBM's BHAG to reshape the computer industry in the early 1960s. To attain this BHAG, IBM put itself at risk by making an all-or-nothing investment in a new computer called the IBM 360. At the time, the 360 was the largest privately financed commercial project ever undertaken; it required more resources than the United States spent on the Manhattan Project to develop the first atomic bomb. *Fortune* magazine called the 360 "IBM's $5,000,000,000 gamble . . . perhaps the riskiest business judgment of recent times." During the 360 introduction, IBM built up nearly *$600 million* of work-in-process inventory and almost needed emergency loans to meet payroll.

EXHIBIT 24

Big Blue Marks Birth of Big Iron

By Daniel Terdiman

02:00 AM Apr. 09, 2004 PT

MOUNTAIN VIEW, California -- The mainframe computer turned 40 years old Wednesday, and hundreds of industry luminaries got together to fete its birthday.

At a celebration of the 40th anniversary of IBM's System/360 at the Computer History Museum in Mountain View, California, the venerable machine was at the center of a Big Blue love fest.

"As I look back on those days, while the making of a revolutionary product was one thing, what was huge was that System/360 was a platform for genuine innovation," Nicholas Donofrio, IBM's senior vice president for technology and manufacturing, told an adoring crowd at the event. "It was a defining moment in the information technology industry. You could say that System/360 was innovating by anticipating the future."

System/360 was the world's first mainframe, and its introduction opened the door to a new era of computing in which, for the first time, it was possible to perform a million instructions per second. Even the lower-end models in the System/360 line were capable of 75,000 instructions a second.

And this newfound power, at prices most sizable businesses could afford, helped the banking and insurance industries, not to mention many others, move into a more modern age.

Indeed, author James Collins has written that the System/360, along with Ford's Model T and the Boeing 707 jet, were the three most important business innovations of all time.

What has set System/360 apart from most other computing projects in history is that its architecture is still alive and well. To prove the point, in conjunction with the 40th-anniversary celebrations, IBM announced a new mainframe computer, its eServer zSeries 890. And while the new 890 series machines have little in common with the System/360 family, they do share one thing: compatibility.

IBM's System/360 team designed the line to be upward- and downward-compatible. That means, says Bob Evans, who first proposed and then headed up the System/360 project in 1964, that work done on just about any model in the 360 family could be ported to another.

"Banks would assess the applications they had to run and make the decision about the computing power they needed," he says. "The beauty of it was that, in most cases, as their business needs grew over time, they could just move to a bigger machine, and run all their applications on" it.

And allowing companies to keep moving their applications forward as their business requirements grew and technology improved -- all without abandoning the System/360 architecture -- has ensured the survival of the line for far longer than anyone ever expected.

"I'm pleased with the endurance of the instruction set and the architecture of the 360," says Evans. "Usually, (computers) last about five years, 10 at the most, then they disappear. You can look over the history of computers, and very few have lasted more than 10 years."

Fred Brooks, the general product manager on the System/360 project, agrees.

"We wrote a paper in 1965 that predicted that the architecture would be good for 25 years," he remembers, "and we're pleased to see that it's going for 40."

Evans enjoys comparing the original System/360 machines with today's personal computers.

"I have a (laptop) that runs 2.8 (million instructions per second) and I paid $2,500 for it, but I got some bells and whistles in the software," he says. "Back in those days, and this is really amazing, if you bought the big machine, you paid ... probably $3 million or $2.5 million.... Now you can buy a machine for one-one-thousandth of that price, and I have a machine that is three times faster."

At the birthday party for the System/360 Wednesday, a genuine enthusiasm for the mainframe was evident. Nearly everyone on hand had gray hair, and those that didn't seemed to wish they could have been part of the computing revolution that the 360 team had begun.

And when Evans and Brooks spoke to their congregation, they could do no wrong.

"I want to thank you for coming tonight to listen to some old geezers talk about old geezing," Brooks began.

In fact, both Brooks and Evans were light on their feet, and kept the crowd loose by mixing humor into what might have otherwise been a dry hour-and-a-half chronicling of the birth of System/360. Laughter at inside jokes far over the heads of laymen in the room was frequent.

Earlier in the evening, at a reception for the System/360 team, it was all geek talk of the highest level. Programmers and engineers from throughout Silicon Valley mingled, networked and expressed their admiration for what IBM had done 40 years ago.

To some, however, the best part of the System/360 legacy was how mainframes have evolved.

Gil Amelio, the former CEO of both National Semiconductor and Apple Computer, said he appreciated how far mainframes have come since they were strictly the province of the "high priests" in corporations' computing departments.

"Now they have brought that power to everyone," Amelio says. "It's testimony to what they've accomplished" with computing.

Amelio also noted that while there will "always be a need for big, clunky computers," mainframes are losing their importance in the business world.

But you wouldn't have known it Wednesday night at the Computer History Museum.

View Photographs

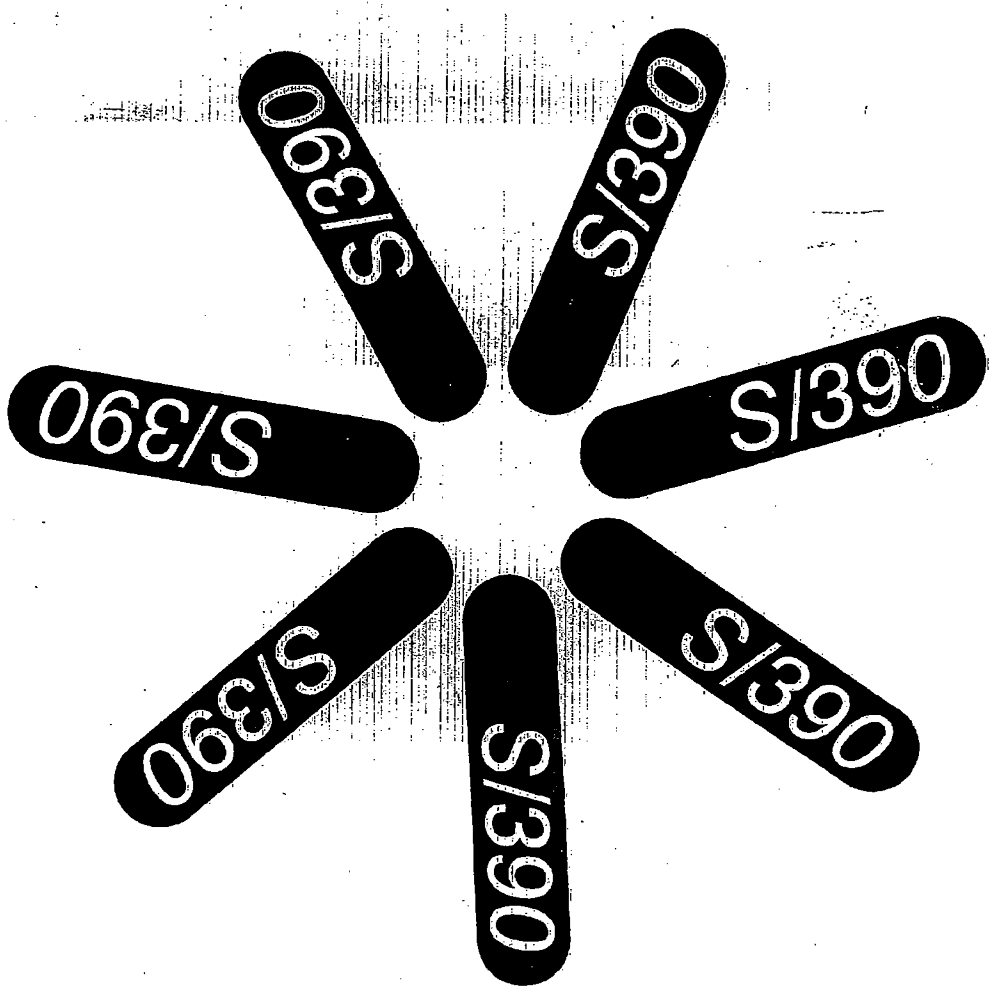
Back to WhatYouSay.net

EXHIBIT 25

SA22-7201-02

Enterprise Systems Architecture/390

Principles of Operation



IBM

Enterprise Systems Architecture/390

SA22-7201-02

Principles of Operation

Note:

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xv.

Third Edition (December 1994)

This edition obsoletes and replaces *Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201-01.

This publication is provided for use in conjunction with other relevant IBM publications, and IBM makes no warranty, express or implied, about its completeness or accuracy. The information in this publication is current as of its publication date but is subject to change without notice.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to:

IBM Corporation
Enterprise Systems Central Architecture
Department E57
522 South Road
POUGHKEEPSIE NY 12601-5400
U.S.A.

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

the physical equipment, whether in the configuration or not, is referred to as the installation.

Model-dependent reconfiguration controls may be provided to change the amount of main and expanded storage and the number of CPUs and channel paths in the configuration. In some instances, the reconfiguration controls may be used to partition a single configuration into multiple configurations. Each of the configurations so reconfigured has the same structure, that is, main and expanded storage, one or more CPUs, and one or more subchannels and channel paths in the channel subsystem.

Each configuration is isolated in that the main and expanded storage in one configuration is not directly addressable by the CPUs and the channel subsystem of another configuration. It is, however, possible for one configuration to communicate with another by means of shared I/O devices or a channel-to-channel adapter. At any one time, the storage, CPUs, subchannels, and channel paths connected together in a system are referred to as being in the configuration. Each CPU, subchannel, channel path, main-storage location, and expanded-storage location can be in only one configuration at a time.

Main Storage

Main storage, which is directly addressable, provides for high-speed processing of data by the CPUs and the channel subsystem. Both data and programs must be loaded into main storage from input devices before they can be processed. The amount of main storage available on the system depends on the model, and, depending on the model, the amount in the configuration may be under control of model-dependent configuration controls. The storage is available in multiples of 4K-byte blocks. At any instant, the channel subsystem and all CPUs in the configuration have access to the same blocks of storage and refer to a particular block of main-storage locations by using the same absolute address.

Main storage may include a faster-access buffer storage, sometimes called a cache. Each CPU may have an associated cache. The effects, except on performance, of the physical construction and the use of distinct storage media are not observable by the program.

Expanded Storage

Expanded storage may be available on some models. Expanded storage, when available, can be accessed by all CPUs in the configuration by means of instructions that transfer 4K-byte blocks of data from expanded storage to main storage or from main storage to expanded storage. These instructions are not described. Another capability for accessing expanded storage is described in the definition of the MOVE PAGE instruction in Chapter 7, "General Instructions," and Chapter 10, "Control Instructions."

Each 4K-byte block in expanded storage is addressed by means of a 32-bit unsigned binary integer called an expanded-storage block number.

Expanded storage is not further described.

CPU

The central processing unit (CPU) is the controlling center of the system. It contains the sequencing and processing facilities for instruction execution, interruption action, timing functions, initial program loading, and other machine-related functions.

The physical implementation of the CPU may differ among models, but the logical function remains the same. The result of executing an instruction is the same for each model, providing that the program complies with the compatibility rules.

The CPU, in executing instructions, can process binary integers and floating-point numbers of fixed length, decimal integers of variable length, and logical information of either fixed or variable length. Processing may be in parallel or in series; the width of the processing elements, the multiplicity of the shifting paths, and the degree of simultaneity in performing the different types of arithmetic differ from one CPU to another without affecting the logical results.

Instructions which the CPU executes fall into five classes: general, decimal, floating-point, control, and I/O instructions. The general instructions are used in performing binary-integer-arithmetic operations and logical, branching, and other

nonarithmetic operations. The decimal instructions operate on data in the decimal format, and the floating-point instructions on data in the floating-point format. The privileged control instructions and the I/O instructions can be executed only when the CPU is in the supervisor state; the semiprivileged control instructions can be executed in the problem state, subject to the appropriate authorization mechanisms.

To perform its functions, the CPU may use a certain amount of internal storage. Although this internal storage may use the same physical storage medium as main storage, it is not considered part of main storage and is not addressable by programs.

The CPU provides registers which are available to programs but do not have addressable representations in main storage. They include the current program-status word (PSW), the general registers, the floating-point registers, the control registers, the access registers, the prefix register, and the registers for the clock comparator and the CPU timer. Each CPU in an installation provides access to a time-of-day (TOD) clock, which may be local to that CPU or shared with other CPUs in the installation. The instruction operation code determines which type of register is to be used in an operation. See Figure 2-2 on page 2-5 for the format of those registers.

PSW

The program-status word (PSW) includes the instruction address, condition code, and other information used to control instruction sequencing and to determine the state of the CPU. The active or controlling PSW is called the current PSW. It governs the program currently being executed.

The CPU has an interruption capability, which permits the CPU to switch rapidly to another program in response to exceptional conditions and external stimuli. When an interruption occurs, the CPU places the current PSW in an assigned storage location, called the old-PSW location, for the particular class of interruption. The CPU fetches a new PSW from a second assigned storage location. This new PSW determines the next program to be executed. When it has finished processing the interruption, the interrupting program may reload the old PSW, making it again the current PSW, so that the interrupted program can continue.

There are six classes of interruption: external, I/O, machine check, program, restart, and supervisor call. Each class has a distinct pair of old-PSW and new-PSW locations permanently assigned in real storage.

General Registers

Instructions may designate information in one or more of 16 general registers. The general registers may be used as base-address registers and index registers in address arithmetic and as accumulators in general arithmetic and logical operations. Each register contains 32 bits. The general registers are identified by the numbers 0-15 and are designated by a four-bit R field in an instruction. Some instructions provide for addressing multiple general registers by having several R fields. For some instructions, the use of a specific general register is implied rather than explicitly designated by an R field of the instruction.

For some operations, two adjacent general registers are coupled, providing a 64-bit format. In these operations, the program must designate an even-numbered register, which contains the leftmost (high-order) 32 bits. The next higher-numbered register contains the rightmost (low-order) 32 bits.

In addition to their use as accumulators in general arithmetic and logical operations, 15 of the 16 general registers are also used as base-address and index registers in address generation. In these cases, the registers are designated by a four-bit B field or X field in an instruction. A value of zero in the B or X field specifies that no base or index is to be applied, and, thus, general register 0 cannot be designated as containing a base address or index.

Floating-Point Registers

Four floating-point registers are available for floating-point operations. They are identified by the numbers 0, 2, 4, and 6 and are designated by a four-bit R field in floating-point instructions. Each floating-point register is 64 bits long and can contain either a short (32-bit) or a long (64-bit) floating-point operand. A short operand occupies the leftmost bit positions of a floating-

EXHIBIT 26

IBM

Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing,* SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

to assign a
subroutine

ted with a
a cataloged
edure con-

outine sub-

ned by its
ven condi-
events that
part of a
bination of
performed
form oper-

cluding at
ssed after

IO system
that may
operation
g wait.

a process

ystem, the
e corona,
ser, oper-
d transfer

on.

m with a
ontrols a

ocess, in
e usually

measures
process
computer
nsforma-
ors. (T)

process
interface

Note: The process interface system may be part of a special-purpose computer.

process entry In ACF/TCAM, a terminal-table entry representing an application program. A process entry must be defined for each queue to which an application program can issue a GET or READ macroinstruction. At least one process entry must be defined for all PUT and WRITE macroinstructions from the same program. See also cascade entry, group entry, line entry, logtype entry, single entry, terminal-table entry.

process exception Synonym for program exception.

process group In the AIX operating system, a grouping of processes that permits the signaling of related groups of processes. A newly created process joins the process group of its creator.

processing (1) The performance of logical operations and calculations on data, including temporary retention of data in processor storage while the data is being operated on. (2) In a document copying machine, the treatment of sensitized material after exposure so as to reveal and retain the image. (T) (3) The action of performing operations on input data.

processing and control element (PCE) In the 8100 Information System, the part of the processor that contains sequencing and processing controls for instruction execution, interruption control, dynamic address translation, and other control and processing functions.

processing intent In IMS/VS, an application program attribute defined in the program communication block (PCB) that specifies the program's database access privileges, such as insert, delete, and replace.

processing level In System/36, a stage in the work-station utility program cycle.

processing limit In IMS/VS, a transaction attribute that defines how many messages the application program can process during one program execution.

processing program (1) A program that performs such functions as compiling, assembling, or translating for a particular programming language. (2) Any program capable of operating in the problem program state. This includes IBM-distributed language processors, application programs, service programs, and user-written programs.

processing services In DPCX, the part of program services that provides data processing capabilities.

processing system See data processing system. (A)

processing unit (1) A functional unit that consists of one or more processors and their internal storages. (I) (A) (2) See primary processing unit, processor. (3) See also mainframe, processor complex. See Figure 117.

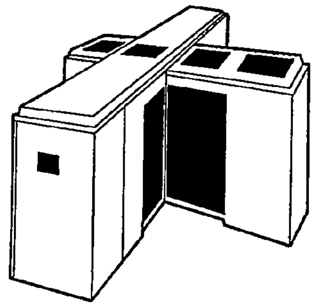


Figure 117. Processing Unit

process interface system A functional unit that adapts process control equipment to the computer system in a process computer system. (T)

process interrupt signal A signal that originates from a technical process and that causes an interrupt in the process computer system. (T)

process lock In the AIX operating system, a lock that allows the calling process to lock or unlock both its text and data segments into memory.

processor (1) In a computer, a functional unit that interprets and executes instructions. A processor consists of at least an instruction control unit and an arithmetic and logic unit. (T) (2) One or more integrated circuits that process coded instructions and perform a task. See also system processor, service processor, input/output processor. (3) Deprecated term for processing program.

processor address space In the IBM 8100 Information System, the set of relocated addresses numbered sequentially from zero to the maximum available address.

processor complex A configuration that consists of all the machines required for operation; for example, an IBM 3081 Processor Complex, consisting of a 3081 Processor Unit, 3082 Processor Controller, 3087 Coolant Distribution Unit, 3089 Power Unit, and a 3278 Display Console Model 2A.

processor configuration In the System/370 computing system, the ability to enable or disable storage elements for one or more processing units. In the multiprocessing mode, any storage element that is enabled for one processing unit can be accessed by

EXHIBIT 27

IBM Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing*, SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

guages, a named group of statements that can be called and evaluated, and can return a value to the calling statement. (7) In Pascal, a routine that is called by coding its name on the right side of an assignment statement. The routine passes a result back to the calling program through the routine name. (8) In PL/I, a procedure that has a RETURNS option in the PROCEDURE statement. A function ends by running a RETURNS statement and returning a scalar value to the point of call. Contrast with subroutine. (9) In SQL, an operation that supplies a single value from another value or from a set of values. A function obtains a single value by applying the function name; for example AVG, to the result of the expression, for example, column-name. See also column function, scalar function. (10) In SMP/E, a system component or licensed program that can be optionally installed in a user's system. Functions are defined to SMP/E by the ++FUNCTION statement. Each function must have a unique function modification identifier (FMID).

functional analysis A systematic investigation of the functions of a real or planned system. (T)

functional design The specification of the functions of the components of a system and of the working relationships among them. (T)

functional diagram A diagram that represents the working relationships among the parts of a system. (A)

functional enhancement package A diskette containing enhancements to DPPX/Base.

functionality Synonym for utility.

functional language A programming language in which computations are expressed in terms of function procedure calls; for example, LISP. (T)

functional macroinstruction A TCAM macroinstruction that performs the operations which are required for messages that are directed to the message handler. Contrast with delimiter macroinstruction.

functional performance Synonym for utility.

functional programming A method for structuring programs mainly as sequences of possibly nested function procedure calls. (T)

functional recovery routine (FRR) In OS/VS, a recovery routine that is used by the locked programs, the service request blocks, and the supervisor control routines.

functional subsystem application (FSA) An area within the Print Services Facility functional subsystem that represents a single printing subsystem and from which the printing subsystem is managed.

functional subsystem (FSS) The Print Services Facility address space created by the job entry subsystem.

functional subsystem interface (FSI) A set of services that allow communication between the JES address space or direct printer services subsystem (DPSS) and the Print Services Facility functional subsystem.

functional test In the 3881 Optical Mark Reader, a diagnostic microprogram used by the microprocessing unit to test overall system reliability.

functional unit An entity of hardware or software, or both, capable of accomplishing a specified purpose. (I) (A)

function authority credential (FAC) In DPPX, a value assigned to each user, each command, and each load module to allow: (a) a user to request that a load module be executed, (b) a user to issue a command, and (c) a load module to request that another load module be executed.

function call An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values.

function check A notification that an unexpected condition has stopped the execution of a program.

function check machine A notification of the malfunction of a machine instruction.

function control sequence (FCS) In the AS/400 system and System/38, a multiple-leaving telecommunications access method (MTAM), control character that controls the flow of individual function streams.

function declaration statement A declaration used to assign a name to a function. (T)

function declarator The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters.

function definition module (1) In DPPX, the programs that control functions performed by an adapter. (2) A module used by DPPX communication programs to access hardware.

EXHIBIT 28

IBM

Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing,* SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

instantiation A formula or pattern rule having its variables replaced by constants. (T)

instant jump A feature of some videodisc players that permits branching between frames within certain minimum distances, usually one to 200 frames away. The branch occurs during the vertical blanking interval between images.

in-stream procedure A set of job control statements placed in the input stream that can be used any number of times during a job by naming the procedure in an execute (EXEC) statement.

instruction (1) A language construct that specifies an operation and identifies its operands, if any. (T) (2) A statement that specifies an operation to be performed by a system and that identifies data involved in the operation. (3) In COBOL and Pascal, one or more clauses, the first of which starts with a keyword that identifies the instruction. Instructions affect the flow of control, provide services to the programmer, or both.

instruction address (1) The address of an instruction word. (I) (A) (2) The address that must be used to fetch an instruction. (A) (3) Contrast with address part.

instruction address register (IAR) (1) A special-purpose register used to hold the address of the next instruction to be executed. Synonymous with program register, instruction pointer register. (T) (2) A register in a processor that contains the address of the next instruction to be performed. (3) See also instruction counter, program counter.

instruction address stop An instruction address that, when fetched, causes execution to stop.

instructional design The field of education that studies the methodology of creating tools, such as computer programs, for enhancing the learning process.

instruction code (1) A code for representing the machine instructions of a computer. (T) (2) See computer instruction code. (A)

instruction control unit In a processing unit, the part that retrieves instructions in proper sequence, interprets each instruction, and applies the proper signals to the arithmetic and logic unit and other parts in accordance with this interpretation. (I) (A)

instruction counter (1) A counter that indicates the location of the next computer instruction to be interpreted. (A) (2) See also instruction address register, program counter.

instruction element (IE) A part of a processor that executes some instructions and generates operand addresses and instruction requests. It also controls the sequencing of instructions through the machine and is usually controlled by microcode.

instruction fetch The act of getting an instruction from storage and loading it into the correct registers.

instruction format The layout of the constituent parts of an instruction. (T)

instruction marker control In dictation equipment, a device used to indicate on an index slip or on the recording medium the position at which an instruction is given. (I)

instruction modifier A word or part of a word that is used to alter an instruction. (I) (A)

instruction pointer In System/38, a pointer that provides addressability for a machine interface instruction in a program.

instruction pointer register Synonym for instruction address register. (T)

instruction register (1) A register used to hold an instruction for interpretation. (I) (A) (2) See control instruction register.

instruction repertoire (1) A complete set of the operators of the statements of a computer programming language, together with a description of the types and meanings that can be attributed to their operands. (A) (2) Loosely, an instruction set. (A)

instructions In SAA Basic Common User Access architecture, text on a panel that tells a user how to interact with a panel and how to continue with the application.

instruction set (1) The set of instructions of a computer, of a programming language, or of the programming languages in a programming system. (I) (A) (2) See computer instruction set.

instruction statement See instruction *(1)*.

instruction time (I-time) The time during which an instruction is fetched from the main storage of a computer into an instruction register. See also execution time.

instruction word A word that represents an instruction. (I) (A)

INT (1) Interior. (2) Internal trace table.

EXHIBIT 29

IBM Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing*, SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

M

M Mega; 1,000,000 in decimal notation. When referring to storage capacity, 2 to the twentieth power; 1,048,576 in decimal notation.

m (1) Milli; one thousandth part. (2) Merge order. (3) Meter.

MAC (1) Medium access control. (2) Message authentication code. (3) Mandatory access control.

MAC frame A transmission frame that controls the operation of the IBM Token-Ring Network and any ring station operations that affect the ring.

machine See accounting machine, electrical accounting machine, Turing machine, universal Turing machine.

machine address Deprecated term for absolute address *(1)*.

machine characteristic A value defined in the computer.

machine check An error condition that is caused by an equipment malfunction.

machine check analysis and recording In VSE, a feature that records machine check error information and then attempts to recover from the error.

machine check handler (MCH) A feature that analyzes errors and attempts recovery by retrying the failing instruction. If retry is unsuccessful, it attempts to correct the malfunction or to isolate the affected task.

machine check interruption (MCI) An interruption that occurs as a result of an equipment malfunction or error.

machine code Synonym for instruction code. (T)

machine configuration record In System/38, a series of data fields, modifiable only by the customer service representative, that describes the hardware.

machine cycle The shortest period of time required to execute an instruction.

machine execution priority In System/38, the priority of a routing step when competing with other routing steps for machine resources.

machine handle In a duplicator, a control for manually cycling the machine. (T) Synonymous with machine handwheel.

machine handwheel Synonym for machine handle.

machine-independent Pertaining to procedures or programs created without regard for the actual devices that are used to process them.

machine information code See system reference code.

machine instruction (1) An instruction that can be directly executed by a processor of a computer. A machine instruction is an element of a machine language. (T) (2) An instruction of a machine language. (3) Synonymous with computer instruction.

machine instruction set Synonym for computer instruction set.

machine interface (MI) (1) The means by which a device, program, user, or system interacts with a computer; for example, commands, instructions, display pointers, menus, keyboards. (2) In System/38, the instruction set and interface to the machine.

machine language (1) An artificial language composed of the machine instructions of a computer. (T) (2) A language that can be used directly by a computer without intermediate processing. (3) Synonymous with computer language.

machine learning The ability of a device to improve its performance based on its past performance. (I) (A)

machine object In the AS/400 system, a program object that has no defined storage form; the object is defined internally to the machine. The machine object is not available to the user. Contrast with data object.

machine operation An elementary function that a computer performs in response to a machine instruction. Synonymous with computer operation. (T)

machine-oriented language Synonym for computer-oriented language.

machine-readable Pertaining to data a machine can acquire or interpret (read) from a storage device, a data medium, or other source.

EXHIBIT 30

IBM

Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing*, SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

ormation about
ınd patches to

ated collection
ıe development

language for

ıtifiable set of
by an assem-
ıutine, or other

ion PRPQ. A
ı alterations or
:s of system
grams. The
ith computing
ıcessing prob-

profile In
ı the Control
thority neces-
:esentative to
l the special
:ontrol rights.

of symbolic

ning environ-
:lopment and
a data proc-
one or more

d System/38,
:nter BASIC
:m from the
ıtements are
ı data mode.

ion Division
:d word that

:ntry in the
:ation Divi-
ıe program-
ıutes to the

ıstem and
:ssifications.
ns that get
ition table.

Program objects are used as the parameter or values of machine instructions. Contrast with system object.

program offering An unwarranted licensed program. See also vendor-logo product.

program operator An VTAM application program that is authorized to issue VTAM operator commands and receive VTAM operator awareness messages. See also solicited message, unsolicited message.

program operator interface (POI) A VTAM function that allows programs to perform VTAM operator functions.

program origin See computer program origin.

program parameter See external program parameter.

program patch (1) A temporary fix that is made to a program. (2) In System/38, a method of repairing a program at the machine interface program template level.

program product Deprecated term for licensed program.

program production time That part of system production time during which a user's computer program is successfully executed. (I) (A)

program recursion level In PL/I, a count that is increased when a program or external procedure is called repeatedly. The program recursion level can be specified on the system debug commands through the RCRLVL parameter. Contrast with procedure recursion level.

program register Synonym for instruction address register. (T) (A)

program required credentials (PRC) In DPPX, the value associated with a load module that determines whether a user must be authorized to request execution of the load module.

program run The performance of one or more programs. (T) (A)

program security See data processing system security.

program selection A function that identifies and actuates a program for operation.

program selector In word processing, a control that allows selection of a machine program for operation. (T)

program-sensitive fault (1) A fault that occurs as a result of the execution of some particular sequence of instructions. (I) (A) (2) Contrast with pattern-sensitive fault.

program services In DPCX, the logical layer that translates data manipulation requests from user programs and system services to resource manipulation requests to a resource manager.

program specification A document that describes the structure and functions of a program in sufficient detail to permit programming and to facilitate maintenance. (T)

program stack (1) In the AS/400 system, a list of programs linked together as a result of programs calling other programs with the CALL instruction, or implicitly from some other event, within the same job. (2) See invocation stack.

program static storage area (PSSA) In the AS/400 system, a system object that contains static variable data for programs on the program stack. The PSSA contains space for program variables that is activated when the program object is activated. The PSSA is contained in the process access group (PAG).

program status register A register that contains conditions that can be tested by branch or jump instructions.

program status vector (PSV) In the IBM 8100 Information System, the formatted information used to control the order in which instructions are executed by the associated program. See primary PSV, secondary PSV.

program status word (PSW) An area in storage used to indicate the order in which instructions are executed, and to hold and indicate the status of the computer system. Synonymous with processor status word.

program structure The manner in which the component parts of a computer program, such as identifiers, declarations, and statements are arranged. (T)

Program Support Representative (PSR) See IBM Program Support Representative.

program table In the AS/400 system, a list of the finance applications for use in an finance job. Each table entry consists of a program ID and the program name and library associated with that ID. Program IDs received in data streams from finance devices are located in the program table to determine the AS/400 application to be called.

EXHIBIT 31

IEEE Std 100-1996

The IEEE Standard Dictionary of Electrical and Electronics Terms

Sixth Edition

Published by the
Institute of Electrical and
Electronics Engineers, Inc.

.ng Language/I

id well-suited to
' type of system
C) 610.13-1993
sferring data be
e program polls
ilable. *Contrast*
ory transfer.
C) 610.10-1994
l method using
ematic sequence
e.
(C) 610.2-1987
and diagnostic
f controlling the
, measurement,
ho prepares se
machine.
(MIL) [2]
r devices that in
f operations in a
) C37.100-1992
ement, and di-
nmands and data
r cards. **(B) (test,**
A device that sets
rforms measure-
, measurement,
compares the re
ammed tolerance
ous other opera
r conditions, are
(MIL) [2]
des the informa
: for a given com
quipment config
mming features,
ssembly features
c manual; instal-
nual; user manual
C) 610.12-1990
aphics System A
omplete graphics
ensional applica-
deling. It was de-
ndards Institute
(C) 610.6-1991
The ordered list
complish a given
ar programming;
) [2], [61], [84]
wer-supply func-
means of an ex-
element. Control
uctances, or vari-
(AE) [41]
on is to establish
C37.100-1981s
larative program-
e form of a rule
lem descriptions
ions for building
C) 610.13-1993
e used to express
uage; query lan-
See also: assem-
urpose program-
ne language.
'90, 610.13-1993
mming language
entific, and busi-
ANSI. *See also:*

ALPHA; block-structured language; FORMAC; IITRAN; LPL. (C) 610.13-1993

programming language API specification The interface between applications and application platforms traditionally associated with programming language specifications, such as program control, math functions, string manipulation, etc. (C/PA) 14252-1996

programming language binding specification For a language-independent specification, a document that specifies, in terms of a particular programming language, the behavior that the language-independent specification specifies in language-independent terms. It may also specify additional behavior that is relevant to the usage of the particular programming language. (C/PA) 1224-1993, 1224.1-1993, 1326.1-1993, 1327-1993, 1328-1993

Programming Language/Data Base (PL/DB) A dialect of PL/1 designed specifically for processing databases and including normal executable statements for arithmetic, conditional, and loop control and supports hierarchical data structures. (C) 610.13-1993

Programming LANguage for Interactive Teaching (PLANIT) An instructional system consisting of a user language that supports the development of computer programs for preparing, editing, and presenting subject matters suitable for interactive presentations. (C) 610.13-1993

programming, linear (1) (general) Optimization problem characterization in which a set of parameter values are to be determined, subject to given linear constraints, optimizing a cost function that is linear in the parameter. *See also:* system. (SMC) [63]
(2) (computers) The analysis or solution of problems in which linear function of a number of variables is to be maximized or minimized when those variables are subject to a number of constraints in the form of linear inequalities. (C) [20], [85]

programming, nonlinear Optimization problem in which any or all of the following are nonlinear in the variables:

a) The objective functions.
b) The defining interrelationships among the variables, the plant description.
c) The constraints.

See also: system. (SMC) [63]

programming, quadratic Optimization problem in which:

a) The objective function is a quadratic function of the variable.
b) The plant description is linear.

See also: system. (SMC) [63]

programming relay A relay whose function is to establish or detect electrical sequences. (PE/SWG) C37.100-1992, C37.90-1978s

programming speed (power supplies) Describes the time requires to change the output voltage of a power supply from one value to another. The output voltage must change across the load and because the supply's filter capacitor forms a resistance-capacitance network with the load and internal source resistance, programming speed can only be described as a function of load. Programming speed is the same as the recovery-time specification for current-regulated operation; it is not related to the recovery-time specification for voltage-regulated operation. (AE) [41]

programming support environment (software) An integrated collection of software tools accessed via a single command language to provide programming support capabilities throughout the software life cycle. The environment typically includes tools for specifying, designing, editing, compiling, loading, testing, configuration management, and project management. Sometimes called integrated programming support environment. *See also:* scaffolding. (C) 610.12-1990

programming system A set of programming languages and the support software (editors, compilers, linkers, etc.) necessary for using these languages with a given computer system. (C) 610.12-1990

program mutation (A) (software) A computer program that has been purposely altered from the intended version to evaluate the ability of test cases to detect the alteration. *See also:* mutation testing. **(B) (software)** The process of creating an altered program as in definition "A." (C) 610.12-1990

program network chart A diagram that shows the relationship between two or more computer programs. (C) 610.12-1990

program production time That part of system production time during which a computer program is successfully executed. (C) 610.10-1994

program protection (software) The application of internal or external controls to preclude any unauthorized access or modification to a computer program. *See also:* computer program; modification. (C/SE) 729-1983s

program register *See:* instruction address register.

program-sensitive fault (1) (computers) A fault that appears in response to some particular sequence of program steps. (C) [20], [85]
(2) (software) A fault that causes a failure when some particular sequence of program steps is executed. *Contrast:* data-sensitive fault. (C) 610.12-1990

program specification (software) Any specification for a computer. program. *Synonym:* design specification. *See also:* design specification; functional specification; performance specification; requirements specification. (C/SE) 729-1983s

program status word (PSW) (A) A computer word that contains information specifying the current status of a computer program. The information may include error indicators, the address of the next instruction to be executed, currently enabled interrupts, and so on. **(B)** A special-purpose register that contains a program status word as in (A). (C) 610.12-1990

program stop (numerically controlled machines) A miscellaneous function command to stop the spindle, coolant, and feed after completion of other commands in the block. It is necessary for the operator to push a button in order to continue with the remainder of the program. (IA) [61], [84]

program support library *See:* software development library.

program structure diagram *See:* structure chart.

program synthesis (software) The use of software tools to aid in the transformation of a program specification into a program that realizes that specification. (C) 610.12-1990

program test time That part of system production time during which a computer program is tested. (C) 610.10-1994

program tracking (communication satellite) A technique for tracking a satellite by pointing a high-gain antenna toward the satellite that employs a computer program for antenna pointing; known orbital parameters are used as an input to the computer program. (COM) [19]

program validation *See:* computer program validation; validation.

progressive grading (telephone switching systems) A grading in which the outlets of different grading groups are connected together in such a way that the number of grading groups connected to each outlet is larger for later choice outlets. (COM) 312-1977w

progressive scanning* *See:* sequential scanning.
* Deprecated.

project (1) (unique identification in power plants) A single- or multiple-unit power plant or major independent related facility. A project is composed of systems and structures and may be defined to include the design, construction, operation, and related activities associated with the project during its life cycle. (PE) 803-1983r
(2) (unique identification in power plants) *See also:* projection. (C) 610.5-1990
(3) A subsystem that is subject to maintenance activity. (C/SE) 1219-1992

EXHIBIT 32

IBM

Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing,* SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

repeatedly producing a display image on a display surface so that the image remains visible. (I) (A) (3) To recharge a memory location in volatile memory with an electric current so that it retains a state or binary value. (4) See system refresh.

Refresh In SAA Basic Common User Access architecture, an action that updates the information that a user is currently looking at.

refreshable The attribute of a load module that prevents it from being modified by itself or by any other module during execution. A refreshable load module can be replaced by a new copy during execution by a recovery management routine without changing either the sequence or results of processing.

refreshable program A program that can be replaced at any time with a new copy without changing either the sequence or results of processing.

refresh rate (1) The number of times per second that a display image is produced for refresh. (I) (A) (2) In word processing, the rate at which a displayed image is renewed in order to appear stable. (T)

regenerate (1) To restore information to its original state. (2) To recharge a dissipating storage device to its fully charged state.

regeneration (1) In computer graphics, the sequence of events needed to generate a display image from its representation in storage. (I) (A) (2) The restoration of stored information. (3) See signal regeneration.

regenerative track Part of a track on a magnetic drum or magnetic disk used in conjunction with a read head and a write head that are connected to function as a circulating storage. (A) Synonymous with revolver track.

region A variable-size subdivision of a dynamic area that is allocated to a job step or a system task.

regional center A control center (class 1 office) connecting sectional centers of the telephone system together. Every pair of regional centers in the United States has a direct circuit group running from one center to the other.

region class In IMS/VS, the class assigned to a message region to indicate the message classes that can be processed within the region. See also message class.

region control task (RCT) In TSO, the control program routine that handles quiesce/restore and LOGON/LOGOFF.

region job pack area (JPA) In OS/VS2, an area in a virtual region that contains modules that are not in the link pack area but are needed for the execution of jobs.

region size The amount of main storage available for a program to run. See also job region, step region.

register (1) A part of internal storage having a specified storage capacity and usually intended for a specific purpose. (T) (2) In DPCX, a field capable of containing 10 digits, a sign, and a decimal point, that can be used by a program for arithmetic calculations and for program control. (3) On a document copying machine, to accurately position the image of the original on the copy material. (T)

registered network ID An 8-byte name included in an IBM-maintained worldwide registry that has a structured format and is assigned to a particular IBM customer to uniquely identify a specific network.

register guides In a document copying machine, indicators on or near the platen that help position the original correctly. (T)

register insertion In a ring network configuration, a ring control scheme in which each station loads the frame it is transmitting into a shift register and, when the ring is idle, inserts the entire contents of the register into the ring. The sending station removes the register contents from the ring when it is returned by the receiving station. See also master node control, slotted-ring control, token-access control.

register length The storage capacity of a register. (I) (A)

registrable resource A logical unit that can be registered with a network node server, a central directory server, or both.

registration (1) The accurate positioning of an entity relative to a reference. (A) (2) On a document copying machine, the consistency of register on successive copies. (T) (3) In a duplicator, the consistency of the relative position of images printed on paper. (T) (4) In X.25, the process used between a data terminal equipment (DTE) and a data circuit-terminating equipment (DCE) to establish an agreement on which optional user facilities will be in effect; for example, the DTE can request that the DCE agree to or stop a previous agreement for an optional user facility. Also, a DCE can indicate the optional user facilities that are available or the optional user facilities that are currently in effect. The negotiation is accomplished through the exchange of registration packets. (5) In the ImagePlus system, the final step in the process of storing an object. Registration represents a positive identification of the object based on the

EXHIBIT 33

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant:	Elko et al.	Confirmation No.: 8602	
Serial No.:	09/337,158	Group Art Unit: 2185	
Filed:	06/21/99	Examiner: Ilwoo Park	

#6/A
7/16/02

Title: METHOD, SYSTEM AND PROGRAM PRODUCTS FOR GENERATING SEQUENCE VALUES THAT ARE UNIQUE ACROSS OPERATING SYSTEM IMAGES

Certificate of Mailing

I hereby certify that this correspondence is being deposited with the U.S. Postal Service as first class mail in an envelope addressed to: Assistant Commissioner for Patents, Washington, D.C. 20231, on June 24, 2002.

Blanche E. Schiller
Blanche E. Schiller
Attorney for Applicants
Reg. No. 35,670

Date of Signature: June 24, 2002

Assistant Commissioner for Patents
Washington, D.C. 20231

Response to Office Action

Dear Sir:

This Response to Office Action is being submitted in reply to the Office Action mailed January 23, 2002 for the above-referenced patent application. A response to the Office Action was initially due on or before April 23, 2002, and thus, a Request For Extension Of Time and the requisite fee are enclosed herewith. Since June 23, 2002 is a Sunday, this Response is being timely filed on Monday, June 24, 2002.

Amendments

Please amend the above-identified application, as follows:

In the Specification:

Please amend page 1, paragraph 2 of the Cross-Reference to Related Applications, as follows:

A1

"METHOD, SYSTEM AND PROGRAM PRODUCTS FOR EMPLOYING EXPANDED PHYSICAL CLOCKS," by Elko et al., Serial No. 09/337,157, (Docket No. PO9-99-064), now US patent No. 6,490,689, filed June 21, 1999. 4/2/04

In the Claims:

Please cancel claim 17, without prejudice, and amend claims 1, 14-16, 18, 20-21, 23, 35-39, 48 and 49, as follows. All of the claims are reproduced for the Examiner's convenience.

Sub B1

A2

1. (AMENDED) A method of generating unique sequence values usable within a computing environment, said method comprising:

providing as one part of a sequence value timing information comprising at least one of time-of-day information and date information, said sequence value representing a timestamp; and

including as another part of said sequence value selected information usable in making said sequence

A2 B1

value unique across a plurality of operating system images on one or more processors of said computing environment.

2. The method of claim 1, further comprising providing as a further part of said sequence value a placeholder value usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information of said sequence value wraps back to zero.

3. The method of claim 1, further comprising providing as a further part of said sequence value a processor identifier.

4. The method of claim 1, wherein said providing and said including are performed by an instruction.

5. The method of claim 4, wherein said providing comprises retrieving, by said instruction, said timing information from a physical clock, and wherein said including comprises retrieving, by said instruction, said selected information from a storage area.

6. The method of claim 5, wherein said storage area is a programmable register set by a set register instruction.

7. The method of claim 6, further comprising initializing said physical clock independently of setting said programmable register.

A

8. The method of claim 4, wherein said instruction is a STORE CLOCK EXTENDED instruction.

9. The method of claim 4, wherein said instruction is issued by a program of said computing environment desiring said sequence value, and wherein said instruction is independent such that communication between said plurality of operating system images is not necessary to generate said sequence value.

10. The method of claim 1, further comprising receiving said timing information from a physical clock of said computing environment.

11. The method of claim 10, further comprising initializing said physical clock to a predefined value using a set instruction.

12. The method of claim 11, further comprising obtaining said selected information from a programmable register independent from said physical clock and said set instruction.

13. The method of claim 1, further comprising obtaining said selected information from a programmable register set by a set register instruction.

14. (AMENDED) A method of generating sequence values usable within a computing environment, said method comprising:

A3

providing as one part of a monotonically increasing sequence value timing information comprising at least one of time-of-day information and date information; and

including as another part of said monotonically increasing sequence value selected information usable in making said monotonically increasing sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

15. (AMENDED) The method of claim 14, further comprising providing as a further part of said monotonically increasing sequence value placeholder information usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

Sub B2

16. (AMENDED) A memory for storing data, said memory comprising:

a representation of a time-of-day clock, said representation being usable within a computing environment and providing a timestamp, said representation comprising:

a timing component comprising timing information including at least one of time-of-day information and date information; and

a programmable field component comprising selected information to make the timestamp unique

A3 B2

across a plurality of operating system images on one or more processors of said computing environment.

17. CANCEL

Sub B3
A4

18. (AMENDED) The memory of claim 16, wherein said representation further comprises a placeholder component usable in ensuring that said timestamp is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

~~19. The memory of claim 16, wherein said representation further comprises a processor identifier component including processor information.~~

A5

20. (AMENDED) A memory for storing data, said memory comprising:

a representation of a time-of-day clock, said representation being usable within a computing environment and providing a monotonically increasing sequence value, said representation comprising:

a timing component comprising timing information including at least one of time-of-day information and date information; and

a programmable field component comprising selected information to ensure said monotonically increasing sequence value is unique across a

A

A3

plurality of operating system images on one or more processors of said computing environment.

21. (AMENDED) The memory of claim 20, wherein said representation further comprises:

a placeholder component usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

~~22. The memory of claim 21, wherein said representation further comprises a processor component including processor information.~~

Sub A4

A4

23. (AMENDED) A system of generating unique sequence values usable within a computing environment, said system comprising:

means for providing as one part of a sequence value timing information comprising at least one of time-of-day information and date information, said sequence value representing a timestamp; and

means for including as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

24. The system of claim 23, further comprising means for providing as a further part of said sequence value a placeholder value usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information of said sequence value wraps back to zero.

25. The system of claim 23, further comprising means for providing as a further part of said sequence value a processor identifier.

26. The system of claim 23, wherein said means for providing and said means for including comprise an instruction.

27. The system of claim 26, wherein said means for providing comprises means for retrieving, by said instruction, said timing information from a physical clock, and wherein said means for including comprises means for retrieving, by said instruction, said selected information from a storage area.

28. The system of claim 27, wherein said storage area is a programmable register set by a set register instruction.

29. The system of claim 28, further comprising means for initializing said physical clock independently of setting said programmable register.

30. The system of claim 26, wherein said instruction is issued by a program of said computing environment

desiring said sequence value, and wherein said instruction is independent such that communication between said plurality of operating system images is not necessary to generate said sequence value.

31. The system of claim 23, further comprising means for receiving said timing information from a physical clock of said computing environment.

32. The system of claim 31, further comprising means for initializing said physical clock to a predefined value using a set instruction.

33. The system of claim 32, further comprising means for obtaining said selected information from a programmable register independent from said physical clock and said set instruction.

34. The system of claim 23, further comprising means for obtaining said selected information from a programmable register set by a set register instruction.

---

35. (AMENDED) A system of generating sequence values usable within a computing environment, said system comprising:

B

means for providing as one part of a monotonically increasing sequence value timing information comprising at least one of time-of-day information and date information; and

A7

means for including as another part of said monotonically increasing sequence value selected information usable in making said monotonically increasing sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

36. (AMENDED) The system of claim 35, further comprising means for providing as a further part of said monotonically increasing sequence value placeholder information usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

Sub B5

37. (AMENDED) A system of generating unique sequence values usable within a computing environment, said system comprising:

a processor adapted to provide as one part of a sequence value timing information comprising at least one of time-of-day information and date information, said sequence value representing a timestamp; and

said processor being further adapted to provide as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

A1

38. (AMENDED) A system of generating sequence values usable within a computing environment, said system comprising:

a processor adapted to provide as one part of a monotonically increasing sequence value timing information comprising at least one of time-of-day information and date information;

said processor being further adapted to include as another part of said monotonically increasing sequence value selected information usable in making said monotonically increasing sequence value unique across a plurality of operating system images on one or more processors of said computing environment; and

said processor being further adapted to provide as another part of said monotonically increasing sequence value placeholder information usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

Sub B6

39. (AMENDED) An article of manufacture, comprising:

at least one computer usable medium having computer readable program code means embodied therein for causing the generating of unique sequence values usable within a computing environment, the computer readable program code means in said article of manufacture comprising:

A7
B6

computer readable program code means for causing a computer to provide as one part of a sequence value timing information comprising at least one of time-of-day information and date information, said sequence value representing a timestamp; and

computer readable program code means for causing a computer to include as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

40. The article of manufacture of claim 39, further comprising computer readable program code means for causing a computer to provide as a further part of said sequence value a placeholder value usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information of said sequence value wraps back to zero.

41. The article of manufacture of claim 39, further comprising computer readable program code means for causing a computer to provide as a further part of said sequence value a processor identifier.

42. The article of manufacture of claim 39, wherein said computer readable program code means for causing a computer to provide comprises computer readable program code means for causing a computer to retrieve, by an instruction, said timing information from a physical clock, and wherein

A

said computer readable program code means for causing a computer to include comprises computer readable program code means for causing a computer to retrieve, by said instruction, said selected information from a storage area.

43. The article of manufacture of claim 42, wherein said storage area is a programmable register set by a set register instruction.

44. The article of manufacture of claim 43, further comprising computer readable program code means for causing a computer to initialize said physical clock independently of setting said programmable register.

45. The article of manufacture of claim 39, further comprising computer readable program code means for causing a computer to receive said timing information from a physical clock of said computing environment.

46. The article of manufacture of claim 45, further comprising computer readable program code means for causing a computer to initialize said physical clock to a predefined value using a set instruction.

47. The article of manufacture of claim 46, further comprising computer readable program code means for causing a computer to obtain said selected information from a programmable register independent from said physical clock and said set instruction.

48. (AMENDED) At least one program storage device readable by a machine, tangibly embodying at least one

A8

program of instructions executable by the machine to perform a method of generating sequence values usable within a computing environment, said method comprising:

providing as one part of a monotonically increasing sequence value timing information comprising at least one of time-of-day information and date information; and

including as another part of said monotonically increasing sequence value selected information usable in making said monotonically increasing sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

49. (AMENDED) The at least one program storage device of claim 48, wherein said method further comprises providing as a further part of said monotonically increasing sequence value placeholder information usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

Remarks

Entry of the amendments, reconsideration of the application, as amended, and allowance of all pending claims are respectfully requested. Claims 1-16 and 18-49 remain pending. Support for the amendments can be found throughout the specification (e.g., pages 28-32). Thus, no new matter is being added.

In accordance with 37 C.F.R. 1.121, a marked-up version of the amended specification and claims is provided on one or more pages separate from the amendment. These pages are appended at the end of the Response.

In the Office Action dated January 23, 2002, claims 1, 4-13, 16, 17, 23, 26-34, 37, 39, and 42-47 are rejected under 35 U.S.C. 102(b) as being anticipated by Frey et al. (U.S. Patent No. 5,416,921); and claims 1, 3, 4, 16, 19, 23, 25, 26, 37, 39 and 41 are rejected under 35 U.S.C. 102(b) as being anticipated by Cwiakala et al. (U.S. Patent No. 5,257,379). Additionally, claims 2, 14, 15, 18, 20, 21, 24, 35, 36, 38, 40, 48 and 49 are rejected under 35 U.S.C. 103(a) as being unpatentable over Frey and Sugiyama (U.S. Patent No. 5,933,625); and claim 22 is rejected under 35 U.S.C. 103(a) as being unpatentable over Frey and Sugiyama and further in view of Cwiakala. Applicants respectfully, but most strenuously, traverse these rejections for the reasons below.

In one aspect, applicants' invention is directed to generating timestamps that are unique across multiple operating system images. The timestamps are monotonically

increasing in value in that two different timestamps resulting from two instructions either on the same CPU or different CPUs correctly imply the sequence of execution of the two instructions.

As one example, applicants claim a method of generating unique sequence values usable within a computing environment (e.g., claim 1). The method includes, for instance, providing as one part of a sequence value timing information comprising at least one of time-of-day information and date information, the sequence value representing a timestamp; and including as another part of the sequence value, selected information usable in making the sequence value unique across a plurality of operating system images on one or more processors of the computing environment. Thus, in one aspect of applicants' claimed invention, a timestamp is generated that is unique across a plurality of operating system images. This is very different from the teachings of any of the references, either alone or in combination.

For example, while Frey describes a technique for creating a unique value across multiple operating system images, this unique value is not and cannot represent a timestamp. In Frey, information is taken from a timestamp in order to make the unique value; however, the unique value in and of itself is no longer a timestamp. That is, the value generated in Frey is not a timestamp. There is no discussion at all in Frey of maintaining the generated value as an increasing sequence value, which is needed for a timestamp. Further, the new value does not have a well-defined anchor bit (i.e., a bit that represents the time value, such as a second, microsecond, etc.) needed for a

timestamp. Thus, the created value of Frey cannot represent a timestamp. Therefore, Frey does not describe, teach or suggest a timestamp that is unique across multiple operating system images, as claimed by applicants.

Similarly, Cwiakala also fails to describe, teach, or suggest a timestamp that is unique across operating system images. Again, while Cwiakala describes using information generated from a clock to create a token, the token is not usable as a timestamp. In Cwiakala, like Frey, the value is created by combining a number of different fields, one of which includes time information; however, the amalgamation of the different fields results in a value that is no longer usable as a timestamp. For example, there is no discussion in Cwiakala (or Frey) of ensuring that the created value is an increasing sequence value, as needed in a timestamp. There is no discussion in Cwiakala of creating a value that represents a time value, as claimed by applicants. Thus, Cwiakala also does not describe, teach or suggest a timestamp that is unique across multiple operating system images. Therefore, Cwiakala does not anticipate applicants' claimed invention.

Based on the foregoing, applicants respectfully submit that independent claim 1, as well as independent claims 16, 23, 37 and 39 are patentable over Frey and Cwiakala. Additionally, the claims that depend from those independent claims are patentable for the same reasons as the independent claims, as well as for their own additional features.

For example, dependent claim 4 recites that the providing and the including elements of independent claim 1 are performed by an instruction. That is, in one aspect of applicants' claimed invention, the timestamp is generated by an instruction (e.g., a Store Clock Extended instruction). This is very different from the teachings of either Frey or Cwiakala, in which the value that is generated in those references is generated by programs combining various fields. While one of those fields is provided using an instruction (e.g., a Store Clock instruction), the other fields used to create the actual value and the combination of the fields are performed by programs and not instructions, as claimed by applicants. Thus, applicants' respectfully request withdrawal of the rejection of claim 4, as well as similar claims.

In addition to the above, it is respectfully submitted that independent claims 14, 20, 35, 38 and 48, which are rejected as being obvious in view of Frey and Sugiyama, are patentable over the teachings of Frey and Sugiyama, either alone or in combination. As one example, independent claim 14 recites a method of generating sequence values usable within a computing environment. The method includes, for instance, providing as one part of a monotonically increasing sequence value timing information comprising at least one of time-of-day information and date information; and including as another part of the monotonically increasing sequence value selected information usable in making the monotonically increasing sequence value unique across a plurality of operating system images on one or more processors of the computing environment. Thus, in one aspect of applicants' claimed invention, a monotonically

increasing sequence value that is unique across a plurality of operating system images is being claimed.

As described above, Frey fails to describe, teach, or suggest monotonically increasing sequence values. Again, while Frey uses a portion of a timestamp to create a value, there is no discussion in Frey that the resulting value is to be an increasing sequence value. Since Frey does not describe, teach, or suggest that the value is to be an increasing sequence value, Frey does not teach or suggest applicants' claimed invention.

Further, Sugiyama does not teach or suggest applicants' claimed invention. While Sugiyama describes a timestamp, Sugiyama makes no mention whatsoever of uniqueness across different operating system images. Thus, Sugiyama does not teach or suggest applicants' claimed invention.

The combination of Frey and Sugiyama also does not teach or suggest applicants' claimed invention. In combination, the references teach, at most, the use of timestamps without regard to uniqueness across multiple operating system images; and the generation of values that are unique across operating system images, but are not monotonically increasing sequence values (or timestamps). There is no discussion, teaching or suggestion in the combination of generating monotonically increasing sequence values that are unique across multiple operating system images. That is, there is no teaching or suggestion in the references of how one would create an increasing sequence value that is unique across operating system images. This is missing from the teachings of the references. Thus, the

combination of Frey and Sugiyama fails to teach or suggest applicants' claimed invention.

Based on the foregoing, applicants respectfully submit that independent claims 14, 20, 35, 38 and 48 are patentable over the combination of Frey and Sugiyama. Further, the dependent claims are patentable for the same reasons as the independent claims, as well as for their own additional features.

Based on the foregoing, applicants respectfully submit that all pending claims are patentable over the references, either alone or in combination.

Should the Examiner wish to discuss this case with applicants' attorney, please contact applicants' attorney at the below listed number.

Respectfully submitted,

Blanche E. Schiller

Blanche E. Schiller
Attorney for Applicants
Registration No. 35,670

Dated: June 24, 2002

HESLIN ROTHENBERG FARLEY & MESITI P.C.
5 Columbia Circle
Albany, New York 12203
Telephone: (518) 452-5600
Facsimile: (518) 452-5579

Version with markings to show changes made

In the Specification:

On page 1, paragraph 2 of the Cross-Reference to Related Applications has been amended, as follows:

"METHOD, SYSTEM AND PROGRAM PRODUCTS FOR EMPLOYING EXPANDED PHYSICAL CLOCKS," by Elko et al., Serial No. [____________] 09/337,157, (Docket No. PO9-99-064), filed June 21, 1999.

In the Claims:

Claims 1, 14-16, 18, 20-21, 23, 35-39, 48 and 49 have been amended, as follows:

1. (AMENDED) A method of generating unique sequence values usable within a computing environment, said method comprising:

providing as one part of a sequence value timing information comprising at least one of time-of-day information and date information, said sequence value representing a timestamp; and

including as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

14. (AMENDED) A method of generating sequence values usable within a computing environment, said method comprising:

providing as one part of a monotonically increasing sequence value timing information comprising at least one of time-of-day information and date information; and

including as another part of said monotonically increasing sequence value selected information usable in making said monotonically increasing sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

[including as another part of said sequence value placeholder information usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.]

15. (AMENDED) The method of claim 14, further comprising providing as a further part of said monotonically increasing sequence value [selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment.] placeholder information usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

16. (AMENDED) A memory for storing data, said memory comprising:

a representation of a time-of-day clock, said representation being usable within a computing environment and providing a timestamp, said representation comprising:

a timing component comprising timing information including at least one of time-of-day information and date information; and

a programmable field component comprising selected information to [provide from said representation a sequence value that is] make the timestamp unique across a plurality of operating system images on one or more processors of said computing environment.

18. (AMENDED) The memory of claim 16, wherein said representation further comprises a placeholder component usable in ensuring that said [sequence value] timestamp is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

20. (AMENDED) A memory for storing data, said memory comprising:

a representation of a time-of-day clock, said representation being usable within [in] a computing environment and providing a monotonically increasing sequence value, said representation comprising:

a timing component comprising timing information including at least one of time-of-day information and date information[, said timing information being at least a part of a value resulting from said representation]; and

a programmable field component comprising selected information to ensure said monotonically increasing sequence value is unique across a plurality of operating system images on one or more processors of said computing environment.

[a placeholder component usable in ensuring that said value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.]

21. (AMENDED) The memory of claim 20, wherein said representation further comprises:

[a programmable field component comprising selected information to ensure said value is unique across a plurality of operating system images on one or more processors of said computing environment.]

a placeholder component usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

23. (AMENDED) A system of generating unique sequence values usable within a computing environment, said system comprising:

means for providing as one part of a sequence value timing information comprising at least one of time-of-day information and date information, said sequence value representing a timestamp; and

means for including as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

35. (AMENDED) A system of generating sequence values usable within a computing environment, said system comprising:

means for providing as one part of a monotonically increasing sequence value timing information comprising at least one of time-of-day information and date information; and

means for including as another part of said monotonically increasing sequence value selected information usable in making said monotonically increasing sequence value unique across a plurality of operating system images on one or more processors of said computing environment [including as another part of said sequence value placeholder information usable in ensuring that said sequence value is an increasing

sequence value, even when a physical clock used to provide said timing information wraps back to zero].

36. (AMENDED) The system of claim 35, further comprising means for providing as a further part of said monotonically increasing sequence value placeholder information usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero [selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment].

37. (AMENDED) A system of generating unique sequence values usable within a computing environment, said system comprising:

a processor adapted to provide as one part of a sequence value timing information comprising at least one of time-of-day information and date information, said sequence value representing a timestamp; and

said processor being further adapted to provide as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

38. (AMENDED) A system of generating sequence values usable within a computing environment, said system comprising:

a processor adapted to provide as one part of a monotonically increasing sequence value timing information comprising at least one of time-of-day information and date information; [and]

said processor being further adapted to include as another part of said monotonically increasing sequence value selected information usable in making said monotonically increasing sequence value unique across a plurality of operating system images on one or more processors of said computing environment; and

said processor being further adapted to provide as another part of said monotonically increasing sequence value placeholder information usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

39. (AMENDED) An article of manufacture, comprising:

at least one computer usable medium having computer readable program code means embodied therein for causing the generating of unique sequence values usable within a computing environment, the computer readable program code means in said article of manufacture comprising:

computer readable program code means for causing a computer to provide as one part of a sequence value timing information comprising at

least one of time-of-day information and date information, said sequence value representing a timestamp; and

computer readable program code means for causing a computer to include as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment.

48. (AMENDED) At least one program storage device readable by a machine, tangibly embodying at least one program of instructions executable by the machine to perform a method of generating sequence values usable within a computing environment, said method comprising:

providing as one part of a monotonically increasing sequence value timing information comprising at least one of time-of-day information and date information; and

including as another part of said monotonically increasing sequence value selected information usable in making said monotonically increasing sequence value unique across a plurality of operating system images on one or more processors of said computing environment [including as another part of said sequence value placeholder information usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero].

49. (AMENDED) The at least one program storage device of claim 48, wherein said method further comprises providing as a further part of said <u>monotonically increasing</u> sequence value <u>placeholder information usable in ensuring that said monotonically increasing sequence value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero</u> [selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment].

Claim 17 has been canceled.

EXHIBIT 34

#20D
3-16-04

OIPE JC16 MAR 03 2004 PATENT & TRADEMARK OFFICE

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants: Elko et al.
Confirmation No.: 8602

Serial No.: 09/337,158
Group Art Unit: 2182

Filed: 06/21/99
Examiner: Ilwoo Park

Title: METHOD, SYSTEM AND PROGRAM PRODUCTS FOR GENERATING SEQUENCE VALUES THAT ARE UNIQUE ACROSS OPERATING SYSTEM IMAGES

CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the U.S. Postal Service as first class mail in an envelope addressed to: Mail Stop RCE, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450, on March 01, 2004.

Jill K. Becker

Jill K. Becker

RECEIVED
MAR 05 2004
Technology Center 2100

Date of Signature: March 01, 2004

To: Mail Stop RCE
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Preliminary Amendment

Dear Sir:

Prior to examination of the above-referenced application, please amend the application, as follows:

Amendment to the Claims

Kindly cancel claim 50, add new claims 51-53, and amend claims 1, 16, 23, 37, 39, as set forth below. In compliance with the Revised Amendment Format published in the Official Gazette on February 25, 2003, a complete listing of claims is provided herein. The changes in the amended claims are shown by strikethrough (for deleted matter) and underlining (for added matter), with the exception that double brackets are used to indicate deleted matter if strikethrough cannot be easily perceived.

1. (Currently Amended) A method of generating unique sequence values usable within a computing environment, said method comprising:

providing as one part of a sequence value timing information comprising at least one of time-of-day information and date information~~, said sequence value being usable as a current time of day value~~; and

including as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment, wherein said sequence value is usable as a current time of day clock value in real-time processing by one or more processors of the computing environment.

2. (Original) The method of claim 1, further comprising providing as a further part of said sequence value a placeholder value usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information of said sequence value wraps back to zero.

3. (Original) The method of claim 1, further comprising providing as a further part of said sequence value a processor identifier.

4. (Original) The method of claim 1, wherein said providing and said including are performed by an instruction.

5. (Original) The method of claim 4, wherein said providing comprises retrieving, by said instruction, said timing information from a physical clock, and wherein said including comprises retrieving, by said instruction, said selected information from a storage area.

6. (Original) The method of claim 5, wherein said storage area is a programmable register set by a set register instruction.

7. (Original) The method of claim 6, further comprising initializing said physical clock independently of setting said programmable register.

8. (Original) The method of claim 4, wherein said instruction is a STORE CLOCK EXTENDED instruction.

9. (Original) The method of claim 4, wherein said instruction is issued by a program of said computing environment desiring said sequence value, and wherein said instruction is independent such that communication between said plurality of operating system images is not necessary to generate said sequence value.

10. (Original) The method of claim 1, further comprising receiving said timing information from a physical clock of said computing environment.

11. (Original) The method of claim 10, further comprising initializing said physical clock to a predefined value using a set instruction.

12. (Original) The method of claim 11, further comprising obtaining said selected information from a programmable register independent from said physical clock and said set instruction.

13. (Original) The method of claim 1, further comprising obtaining said selected information from a programmable register set by a set register instruction.

14. (Canceled)

15. (Canceled)

17 ~~16~~. (Currently Amended) A memory for storing data, said memory comprising:

a representation of a time-of-day clock, said representation being usable within a computing environment and providing a value usable as a current time of day clock value in real-time processing by one or more processors of the computing environment, said representation comprising:

a timing component comprising timing information including at least one of time-of-day information and date information; and

a programmable field component comprising selected information to make the value unique across a plurality of operating system images on one or more processors of said computing environment.

17. (Canceled)

18. (Previously Presented) The memory of claim 17 ~~16~~, wherein said representation further comprises a placeholder component usable in ensuring that said value is an increasing sequence value, even when a physical clock used to provide said timing information wraps back to zero.

19. (Original) The memory of claim 17 ~~16~~, wherein said representation further comprises a processor identifier component including processor information.

20. (Canceled)

21. (Canceled)

22. (Canceled)

20 ~~23~~. (Currently Amended) A system of generating unique sequence values usable within a computing environment, said system comprising:

means for providing as one part of a sequence value timing information comprising at least one of time-of-day information and date information~~, said sequence value being usable as a current time of day value~~; and

means for including as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment, wherein said sequence value is usable as a current time of day clock value in real-time processing by one or more processors of the computing environment.

D 21 ~~24~~. (Original) The system of claim 20 ~~23~~, further comprising means for providing as a further part of said sequence value a placeholder value usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information of said sequence value wraps back to zero.

22 ~~25~~. (Original) The system of claim 20 ~~23~~, further comprising means for providing as a further part of said sequence value a processor identifier.

23 ~~26~~. (Original) The system of claim 20 ~~23~~, wherein said means for providing and said means for including comprise an instruction.

24 ~~27~~. (Original) The system of claim 23 ~~26~~, wherein said means for providing comprises means for retrieving, by said instruction, said timing information from a physical clock, and wherein said means for including comprises means for retrieving, by said instruction, said selected information from a storage area.

D

25 ~~28~~. (Original) The system of claim 24 ~~27~~, wherein said storage area is a programmable register set by a set register instruction.

26 ~~29~~. (Original) The system of claim 25 ~~28~~, further comprising means for initializing said physical clock independently of setting said programmable register.

27 ~~30~~. (Original) The system of claim 23 ~~26~~, wherein said instruction is issued by a program of said computing environment desiring said sequence value, and wherein said instruction is independent such that communication between said plurality of operating system images is not necessary to generate said sequence value.

28 ~~31~~. (Original) The system of claim 20 ~~23~~, further comprising means for receiving said timing information from a physical clock of said computing environment.

D

29 ~~32~~. (Original) The system of claim 28 ~~31~~, further comprising means for initializing said physical clock to a predefined value using a set instruction.

30 ~~33~~. (Original) The system of claim 29 ~~32~~, further comprising means for obtaining said selected information from a programmable register independent from said physical clock and said set instruction.

31 ~~34~~. (Original) The system of claim 20 ~~23~~, further comprising means for obtaining said selected information from a programmable register set by a set register instruction.

35. (Canceled)

36. (Canceled)

32 ~~37~~. (Currently Amended) A system of generating unique sequence values usable within a computing environment, said system comprising:

D

a processor adapted to provide as one part of a sequence value timing information comprising at least one of time-of-day information and date information~~, said sequence value being usable as a current time of day value~~; and

said processor being further adapted to provide as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment, wherein said sequence value is usable as a current time of day clock value in real-time processing by one or more processors of the computing environment.

38. (Canceled)

33 ~~39~~. (Currently Amended) An article of manufacture, comprising:

at least one computer usable medium having computer readable program code means embodied therein for causing the generating of unique sequence values usable within a computing environment, the computer readable program code means in said article of manufacture comprising:

computer readable program code means for causing a computer to provide as one part of a sequence value timing information comprising at least one of time-of-day information and date information~~, said sequence value being usable as a current time of day value~~; and

computer readable program code means for causing a computer to include as another part of said sequence value selected information usable in making said sequence value unique across a plurality of operating system images on one or more processors of said computing environment, wherein said sequence value is usable as a current time of day clock value in real-time processing by one or more processors of the computing environment.

34 ~~40~~. (Original) The article of manufacture of claim 33 ~~39~~, further comprising computer readable program code means for causing a computer to provide as a further part of said sequence value a placeholder value usable in ensuring that said sequence value is an increasing sequence value, even when a physical clock used to provide said timing information of said sequence value wraps back to zero.

35 ~~41~~. (Original) The article of manufacture of claim 33 ~~39~~, further comprising computer readable program code means for causing a computer to provide as a further part of said sequence value a processor identifier.

36 ~~42~~. (Original) The article of manufacture of claim 33 ~~39~~, wherein said computer readable program code means for causing a computer to provide comprises computer readable program code means for causing a computer to retrieve, by an instruction, said timing information from a physical clock, and wherein said computer readable program code means for causing a computer to include comprises computer readable program code means for causing a computer to retrieve, by said instruction, said selected information from a storage area.

37 ~~43~~. (Original) The article of manufacture of claim 36 ~~42~~, wherein said storage area is a programmable register set by a set register instruction.

38 ~~44~~. (Original) The article of manufacture of claim 37 ~~43~~, further comprising computer readable program code means for causing a computer to initialize said physical clock independently of setting said programmable register.

39 ~~45~~. (Original) The article of manufacture of claim 33 ~~39~~, further comprising computer readable program code means for causing a computer to receive said timing information from a physical clock of said computing environment.

40 ~~46~~. (Original) The article of manufacture of claim 39 ~~45~~, further comprising computer readable program code means for causing a computer to initialize said physical clock to a predefined value using a set instruction.

41 ~~47~~. (Original) The article of manufacture of claim 40 ~~46~~, further comprising computer readable program code means for causing a computer to obtain said selected information from a programmable register independent from said physical clock and said set instruction.

48. (Canceled)

49. (Canceled)

50. (Canceled)

14 ~~51~~. (New) The method of claim 1, wherein the sequence value that is generated is considered as one entity, said one entity being usable as the current time of day clock value.

15 ~~52~~. (New) The method of claim 1, wherein said sequence value indicates a correct sequence of execution of an instruction, regardless of which processor of the one or more processors executes the instruction.

16 ~~53~~. (New) The method of claim 1, wherein the sequence value has a predictable resolution.

Remarks

Entry of this amendment, reconsideration of the application, as amended, and allowance of all pending claims are respectfully requested. Claims 1-13, 16, 18-19, 23-34, 37, 39-47 and 51-53 are pending.

With the above amendments, applicants are clarifying that the sequence value is usable as a current time of day clock value in real-time processing. Further, applicants have added dependent claims that define additional characteristics of the sequence value that is generated. Support for these amendments can be found throughout the specification (e.g., page 2, line 15-page 3, line 17; page 11, lines 9-14; page 15, lines 20-23; page 16, lines 24-29; page 18, lines 1-7; page 21, lines 21-23; page 29, lines 8-13; page 29, lines 10-15; page 31, lines 20-29; page 36). Thus, no new matter is added.

In the Office Action dated September 29, 2003, claims 1, 3-13, 16, 19, 23, 25-34, 37, 39 and 41-47 are rejected under 35 U.S.C. 102(b) as being anticipated by Wanish, IBM Technical Disclosure Bulletin, Vol. 23, No. 8, pages 3819-3820, January, 1981. Further, claims 2, 18, 24 and 40 are rejected under 35 U.S.C. 103(a) as being unpatentable over Wanish in view of Sugiyama (U.S. Patent No. 5,933,625). Applicants respectfully, but most strenuously, traverse these rejections to any extent deemed applicable to the amended claims.

In one aspect, applicants' invention is directed to generating a sequence value that is unique across multiple operating system images and is used as a current time of day clock value by one or more processors of the computing environment. That is, the sequence value, as generated, reflects the time at which time is requested by a processor and the processor can use (i.e., act upon) that value in real time. The sequence value is, for instance, monotonically increasing in that two different values resulting from two instructions either on the same CPU or different CPUs correctly imply the sequence of execution of the two instructions.

As one example, applicants claim a method of generating unique sequence values usable within a computing environment (e.g., claim 1). The method includes, for instance, providing as one part of a sequence value timing information comprising at least one of time of day information and date information; and including as another part of the sequence value selected

information usable in making the sequence value unique across a plurality of operating system images on one or more processors of the computing environment, wherein the sequence value is usable as a current time of day clock value in real-time processing by one or more processors of the computing environment. Thus, in one aspect of applicants' claimed invention, a sequence value is generated that is unique across a plurality of operating system images and is still usable as a current time of day clock value in real-time processing. That is, the resulting sequence value, once generated, still has the property of being able to be used as a time of day value for real-time processing. This is very different from the teachings of Wanish.

Wanish describes the creation of a unique identifier, as opposed to the creation of a unique current time of day value. The identifier in Wanish includes some time information, but the identifier itself is not usable as a current time of day value in real-time processing. This is evident in Wanish by, for instance, the fact that in order to use any time information from the Wanish identifier that information needs to be extracted, since only a portion of the identifier includes the time information, unlike applicants' sequence value in which the entire value is the timing information. The extracting of the information takes time, and thus, cannot be used for real-time processing.

It is indicated in Wanish that the id can be useful in analyzing operating system dumps (Wanish, p. 3820). However, this analysis is in post-processing, not in real-time processing, as claimed by applicants in claim 1, in which it is explicitly recited that the sequence value is usable as a current time of day clock value in real-time processing. Thus, applicants respectfully submit that Wanish does not describe applicants' claimed element of generating a sequence value that is usable as a current time of day clock value in real-time processing.

Further, applicants respectfully submit that the identifier created in Wanish, i.e., the identifier, as a whole, is not used or is not usable as a current time of day value in real-time processing. While information generated from a clock is used to create the identifier in Wanish, the identifier, once created, is not usable as a current time of day value. That is, in Wanish, the identifier is created by combining a number of different fields, one of which includes time information. However, the amalgamation of the different fields results in a value that is no longer usable as a time value. This is because in Wanish there is no requirement that the created

identifier represent the current time of day and that the identifier be usable as a time of day value in real-time processing.

For example, there is no discussion in Wanish of ensuring that the created value is an increasing sequence value, as needed in the time of day value. Again, while a clock itself produces increasing sequence values, once that clock information is combined with other information, there is no requirement in Wanish of maintaining the increasing sequence value in the resulting value, so that it can be used for a current time-of-day value. For instance, assume the technique of Wanish is used to create two ids for two different operating systems. Assume the first id is created on July 8 at 10:00 in the morning and the other is created on July 8 at 10:30 in the morning. To be a time of day value, the second value is to have a greater value than the first. However, there is no such requirement in Wanish, and since information other than time information is also used to create the ids, it is very possible that the ids do not follow in sequence order. Take the following example: using the various information to create the ids, one possible value for the first id is: 123456707081000, and a possible value for the second id is 012345607081030. Thus, while the second id was created after the first, it has a smaller value than the first. Thus, the ids cannot be relied on for current time of day values. There is no requirement in Wanish that the resulting id be usable as a time of day representation. Wanish does not even use the whole clock in its value. Thus, applicants respectfully submit that the ids of Wanish are not usable as time of day values within a computing environment, as claimed by applicants, and thus, Wanish does not anticipate applicants' claimed invention.

Based on the foregoing, applicants respectfully submit that claim 1, as well as the other independent claims, are patentable over Wanish. Further, the dependent claims are patentable for the same reasons as the independent claims, as well as for their own additional features. For example, dependent claim 51, explicitly recites that the sequence value that is generated is considered as one entity and it is that entity that is usable as the current time of day clock value. This is very different from Wanish because the resulting value in Wanish cannot be used as a current time of day clock, as described above. While Wanish includes some time information, that time information is combined with other fields and the resulting value cannot be used as a current time of day clock value. Thus, applicants respectfully submit that claim 51 is patentable over Wanish.

Dependent claims 52 and 53 also describe characteristics of applicants' sequence value that are missing from the generated identifier of Wanish. Thus, applicants respectfully submit that those claims are also patentable over Wanish.

For all of the above reasons, applicants respectfully submit that their invention is patentable over Wanish. Additionally, the other cited art does not overcome the deficiencies of Wanish. Thus, applicants respectfully request an indication of allowance of all pending claims.

Applicants urge the Examiner to contact applicants' representative should the Examiner still have further concerns regarding this application.

Respectfully submitted,

Blanche E. Schiller

Blanche E. Schiller
Attorney for Applicants
Registration No.: 35,670

Dated: March 01, 2004.

HESLIN ROTHENBERG FARLEY & MESITI P.C.
5 Columbia Circle
Albany, New York 12203-5160
Telephone: (518) 452-5600
Facsimile: (518) 452-5579

EXHIBIT 35

Exhibit A

AGREED CONSTRUCTIONS

U.S. Patent No. 6,654,812

'812 Claim Term	Agreed Construction
generating an IP datagram	In an IP packet, combining a source and destination address with a data payload
network	Communications links between the network ports of two or more computer systems

U.S. Patent No. 5,687,106

'106 Claim Term	Agreed Construction
directly converted	converted without undergoing intermediary conversions
floating point architecture	A way of representing floating point numbers in bits, including the rules for performing mathematical operations on those representations
internal floating point format	The format of a floating point number within a floating point unit (FPU) that is used in the execution of the floating point operations.

U.S. Patent No. 5,825,678

'678 Claim Term	Agreed Construction
condition code	a bit or collection of bits that summarizes the results of the execution of an instruction
floating point data class	One or more categories of floating point numbers possessing one or more specified characteristics

U.S. Patent No. 5,696,709

'709 Claim Term	Agreed Construction
default rounding mode	The rounding mode to be used whenever the current floating point instruction does not specify some other mode
selectively override	selecting from a set of options to override a default operation

Exhibit A

U.S. Patent No. 6,775,789

'789 Claim Term	**Agreed Construction**
wraps back to zero	resets to zero once the maximum value is exceeded
programmable register	A register whose value can be set by one or more programs
set register instruction	An instruction that sets a given register to a value
considered as one entity	stored contiguously in memory
processor identifier	collection of bits that uniquely specifies a particular processor

U.S. Patent No. 5,987,495

'495 Claim Term	**Agreed Construction**
a processor is controlled by a program status word	The processor is directed in the execution of a program by the contents of a program status word which indicates the next instruction executed and contains controls constraining the operating state and authority of the program executing under that PSW.
program context	contents of the general registers, the access registers, and the program status word
decoding an instruction	Interpreting the fields of an instruction by a processor
offset	value added to an address to reach the desired address
pointing to a save area	providing the location of an area of main storage in which contents of a program status word and registers are saved

Exhibit A

U.S. Patent No. 5,414,851

'851 Claim Term	Agreed Construction
image identifier	a number assigned to each of the plural OSs executing in different resource partitions of the computer system, or the hypervisor
sharing set	a set of control blocks in which each such control block corresponds to the same input/output resource
input/output resource identifier	a collection of bits that identify an I/O resource
hypervisor	a component that supervises the partitions, and programs within such partitions, of the computer electronic complex

U.S. Patent No. 5,953,520

'520 Claim Term	Agreed Construction
native	pertaining to the architecture on which the emulator runs
guest	pertaining to the architecture that is being emulated
native addresses	addresses specifying native memory locations
native address space	a range of addresses available to the instructions written for the native architecture
guest address space	a range of addresses available to the instructions written for the guest architecture
native physical addresses	addresses of the native physical memory location
a series of guest instructions	a sequence of one or more guest instructions
emulates	using a data processing system to imitate another data processing system, so that the imitating system accepts the same data, executes the same programs, and achieves the same results as the imitated system.
guest real address	an address that would be used by the emulated data processing system to specify a location in its physical memory, if such a physical memory were available
guest logical address	the number used by a guest program to indirectly reference a location in memory
guest instruction storage	a set of locations in memory containing guest instruction
semantic routine storage	a set of locations in memory containing semantic routines
translating said guest logical address into a guest real address	producing a guest real address to which the given guest logical address corresponds
translating said guest real address into a native physical address	producing a native physical address to which the given guest real address corresponds

Exhibit A

U.S. Patent No. 6,009,261

'261 Claim Term	**Agreed Construction**
emulation	using a data processing system to imitate another data processing system, so that the imitating system accepts the same data, executes the same programs, and achieves the same results as the imitated system.
incompatible	pertaining to the architecture that is being emulated
target	pertaining to the architecture on which the emulator runs
interpreting an instruction	decoding the fields of an instruction by a processor
preprocessing each target instruction	copying or modifying some or all of the target instruction before it is executed

EXHIBIT 36

US005732235A

United States Patent [19]

Kahle et al.

[11] Patent Number: 5,732,235

[45] Date of Patent: Mar. 24, 1998

[54] **METHOD AND SYSTEM FOR MINIMIZING THE NUMBER OF CYCLES REQUIRED TO EXECUTE SEMANTIC ROUTINES**

[75] Inventors: **James Allan Kahle; Soummya Mallick; Larry Bryce Phillips; Russell Adley Reininger**, all of Austin, Tex.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **591,291**

[22] Filed: **Jan. 25, 1996**

[51] **Int. Cl.**[6] **G06F 9/30**

[52] **U.S. Cl.** **395/385**; 395/384

[58] **Field of Search** 395/384, 385

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,984,154	1/1991	Hanatani et al. .
5,021,945	6/1991	Morrison et al. .
5,136,697	8/1992	Johnson .
5,283,873	2/1994	Steely, Jr. et al. .
5,333,280	7/1994	Ishikawa et al. .
5,333,283	7/1994	Emma et al. .
5,367,703	11/1994	Levitan .
5,381,532	1/1995	Suzuki .
5,381,533	1/1995	Peleg et al. .

FOREIGN PATENT DOCUMENTS

3171327 7/1991 Japan .

OTHER PUBLICATIONS

IBM Technical Disclosure Bulletin, vol. 33 No. 11, Apr. 1991; pp. 66–69.

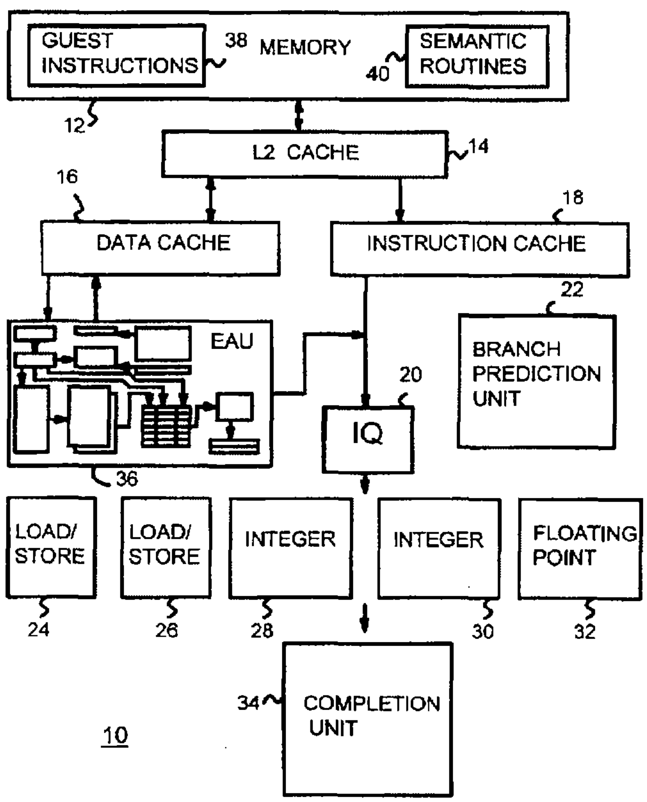
Primary Examiner—Krisna Lim

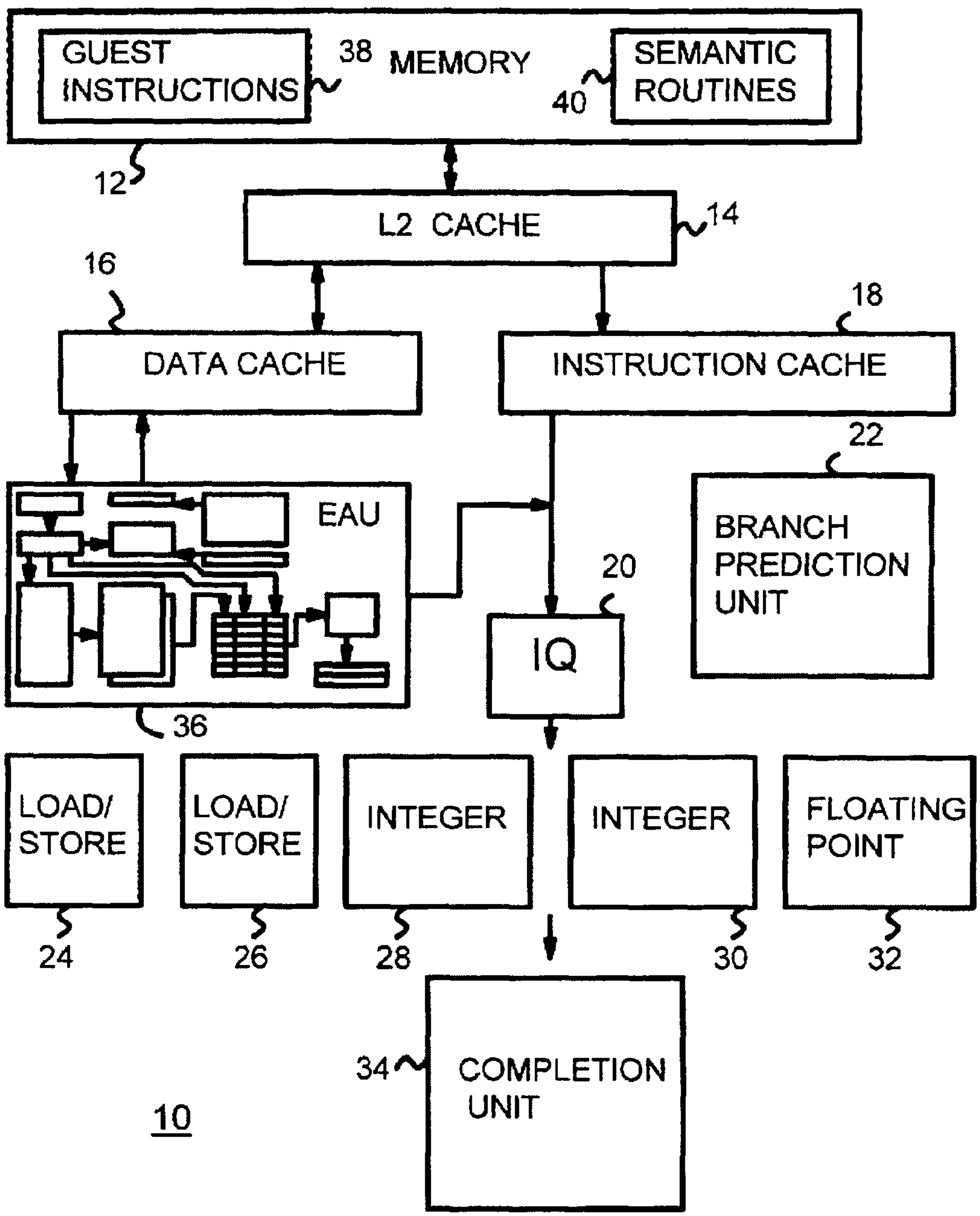
Attorney, Agent, or Firm—Sawyer & Associates; Mark E. McBurney

[57] **ABSTRACT**

A system and method for reducing the cycle time necessary to execute semantic routines in a processor that emulates guest instructions. Each of the semantic routines includes a block of host instructions for performing the function of the corresponding guest instruction, and the last instruction in each of the semantic routines is a branch instruction. The method and system first determines the block length of each of the semantic routines. When a first guest instruction is encountered, the block of instructions in a first semantic routine corresponding to a guest instruction is executed. The block length of first semantic routine is then used to determine when to fetch a second semantic routine without fetching and decoding the branch instruction in the first semantic routine, thereby increasing emulation performance.

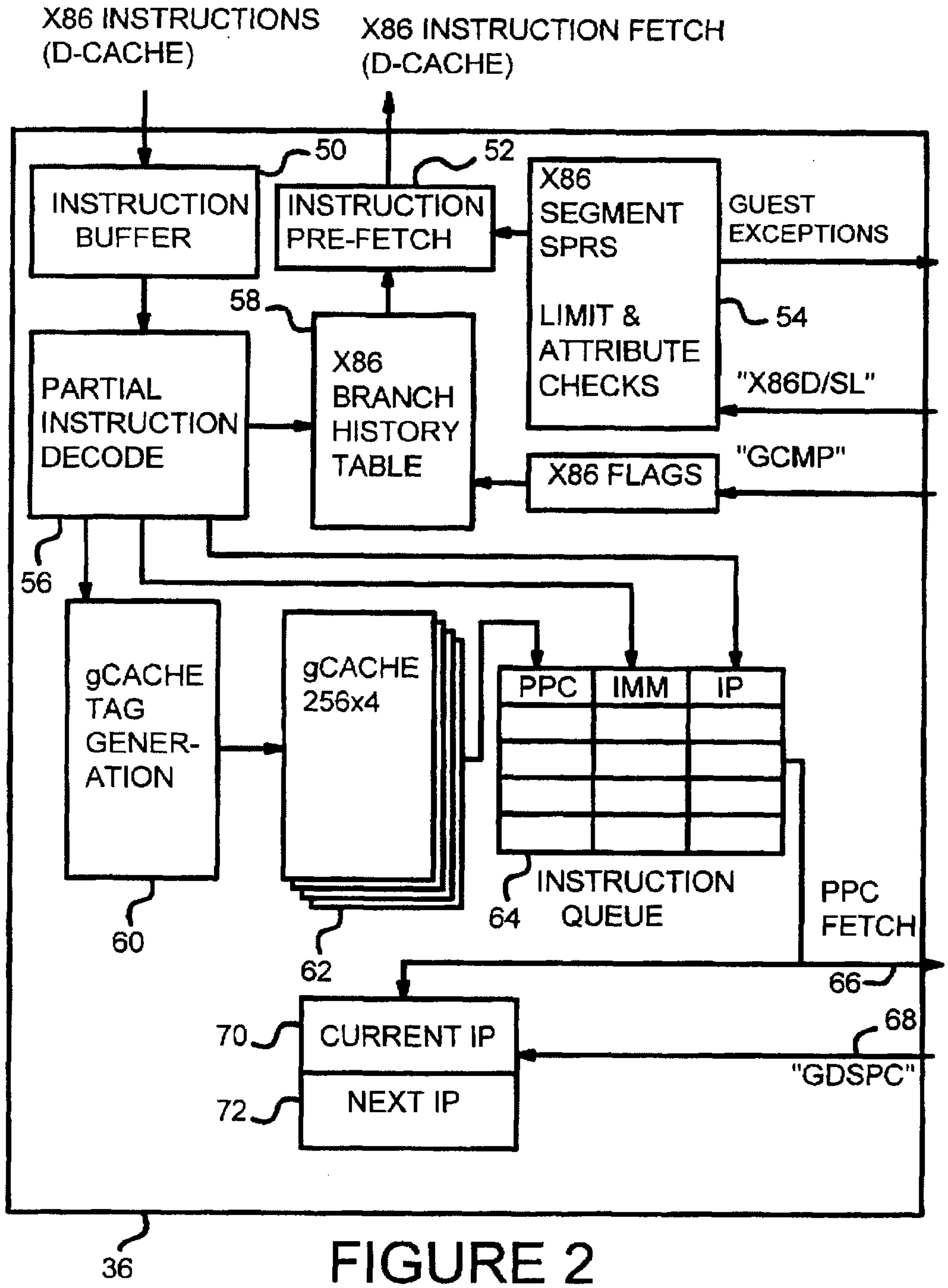
11 Claims, 6 Drawing Sheets





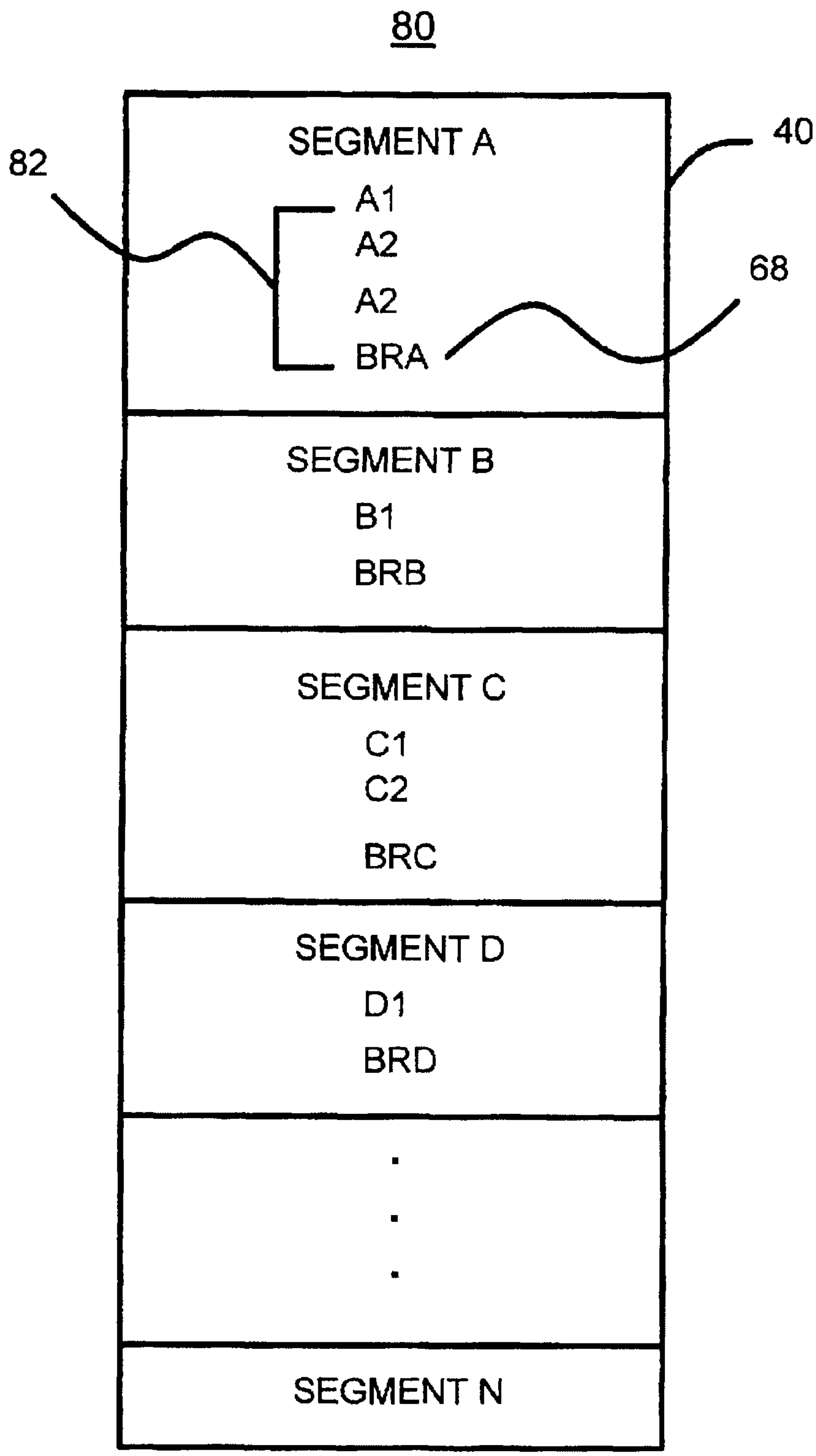
GUEST INSTRUCTIONS
38
MEMORY
40
SEMANTIC ROUTINES
12
L2 CACHE
14
16
18
DATA CACHE
INSTRUCTION CACHE
22
EAU
BRANCH PREDICTION UNIT
20
IQ
36
LOAD/ STORE
LOAD/ STORE
INTEGER
INTEGER
FLOATING POINT
24
26
28
30
32
34
COMPLETION UNIT
10

FIGURE 1



X86 INSTRUCTIONS (D-CACHE)
X86 INSTRUCTION FETCH (D-CACHE)
50
52
INSTRUCTION BUFFER
INSTRUCTION PRE-FETCH
X86 SEGMENT SPRS
LIMIT & ATTRIBUTE CHECKS
GUEST EXCEPTIONS
54
58
X86 BRANCH HISTORY TABLE
PARTIAL INSTRUCTION DECODE
"X86D/SL"
"GCMP"
X86 FLAGS
56
gCACHE TAG GENER-ATION
gCACHE 256x4
PPC
IMM
IP
INSTRUCTION QUEUE
64
60
62
PPC FETCH
66
68
70
CURRENT IP
"GDSPC"
72
NEXT IP
36

FIGURE 2

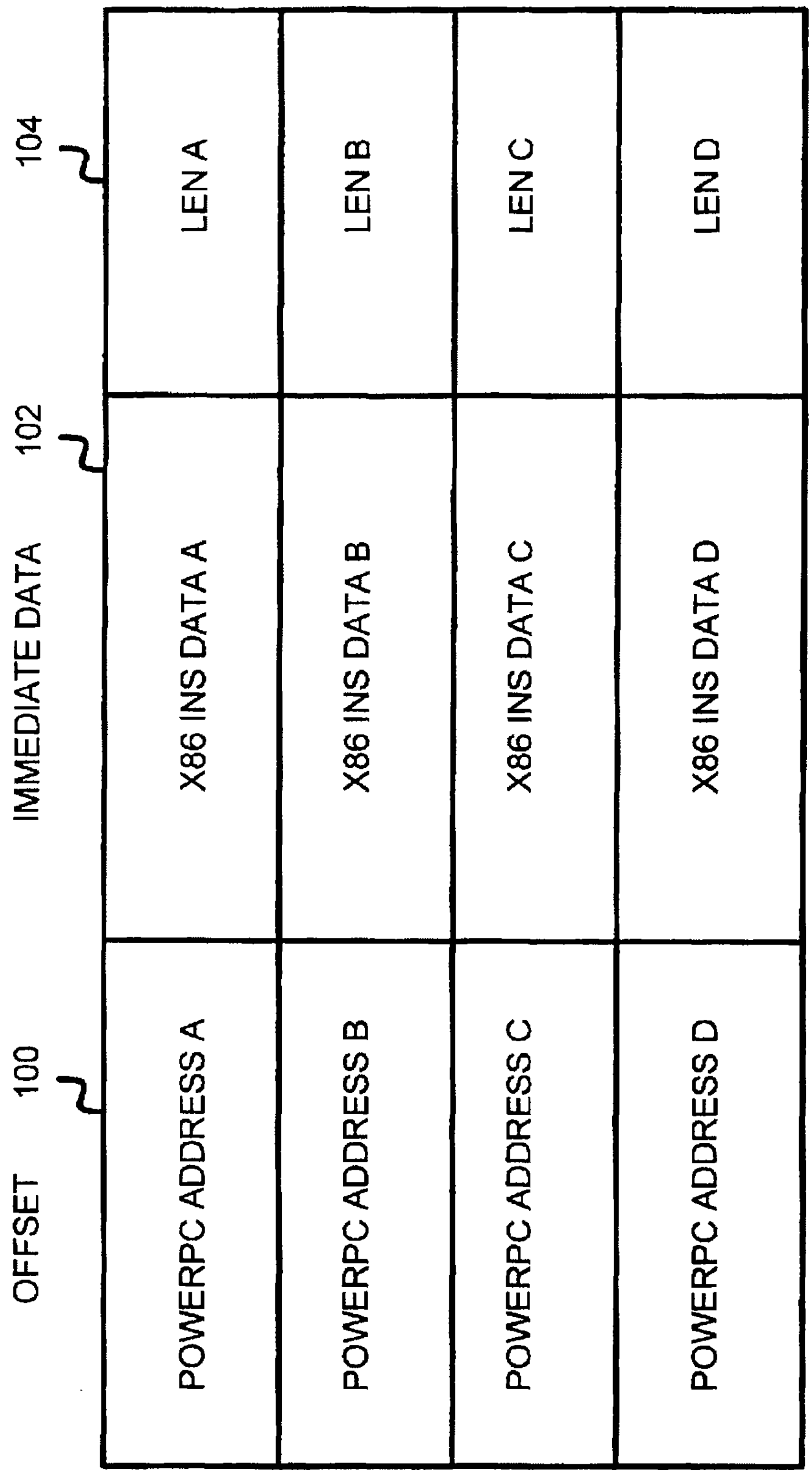


80
82
40
68
SEGMENT A
A1
A2
A2
BRA
SEGMENT B
B1
BRB
SEGMENT C
C1
C2
BRC
SEGMENT D
D1
BRD
SEGMENT N

FIGURE 3

CYCLE	1	2	3	4	5	6	7	8	9	10	11
FETCH	PPC A1 PPC A2	PPC A3 PPC BRA		PPC B1 PPC BRB		PPC C1 PPC C2	PP BRC		PPC D1 PPC BRD		
DEC		PPC A1 PPC A2	PPC A3 PPC BRA		PPC C1 PPC BRB		C1 C2	PP BRC		D1	
EXEC			PPC A1 PPC A2	PC A3		B1		C1 C2			D1

FIGURE 4


OFFSET
100
IMMEDIATE DATA
102
104
POWERPC ADDRESS A
X86 INS DATA A
LEN A
POWERPC ADDRESS B
X86 INS DATA B
LEN B
POWERPC ADDRESS C
X86 INS DATA C
LEN C
POWERPC ADDRESS D
X86 INS DATA D
LEN D

62

FIGURE 5

CYCLE	1	2	3	4	5	6	7	
FETCH	A1 A2	A3 BRA	B1 BRB	C1 C2	D1			
DEC		A1 A2	A3	B1	C1 C2	D1		
EXEC			A1 A2	A3	B1	C1 C2	D1	

FIGURE 6

METHOD AND SYSTEM FOR MINIMIZING THE NUMBER OF CYCLES REQUIRED TO EXECUTE SEMANTIC ROUTINES

FIELD OF THE INVENTION

The present invention relates to a method and system for emulating guest instructions in a host processor, and more particularly to a method and system for minimizing the number of cycles required to execute semantic routines that emulate the guest instructions.

BACKGROUND OF THE INVENTION

There are two methods for emulating in software a guest instruction set, such as the Intel x86 instruction set, on a host processor, such as the Motorola PowerPC™. The first method is known as interpretation, which uses an interpreter, and the second as dynamic translation, which uses a dynamic translator. Interpreters mimic the fetch, decode, and execute cycles performed by hardware in traditional host processors. Dynamic translators fetch and decode a block of guest instructions and translate them into host instructions, which are then reused. Each method has strengths and weaknesses.

Interpreters typically use less memory, but perform slower than dynamic translators. Dynamic translators perform well when the guest program include parts that are frequently reused. But dynamic translators perform poorly when the guest program modifies parts that had previously been translated.

The assignee of the present application has developed hardware that assists software emulation methods in order to obtain a solution that has the positive characteristics of both methods: the small size of the interpreter and the speed of execution of the dynamic translator. Before the assisting hardware is used, the sequence of host instructions necessary to perform the function of each guest instruction are compiled into separate functions, cared semantic routines, and stored in memory. The hardware works in conjunction with software to fetch guest instructions, decode them, and dispatch the corresponding semantic routine in the host processor.

Each of the semantic routines ends with a branch instruction to signal when to dispatch the next semantic routine to be executed. However, the disadvantage of this method is that the each of the branch instructions must first be fetched and decoded before the dispatch of the next semantic routine can occur. This results in a significant processing delay due to the addition of extra processor cycles.

What is needed therefore is a system and method for reducing the cycle time necessary to execute a semantic routine. The present invention addresses such a need.

SUMMARY OF THE INVENTION

The present invention provides a method and system for reducing the cycle time necessary to execute semantic routines in a processor that emulates guest instructions. Each of the semantic routines includes a block of host instructions for performing the function of the corresponding guest instruction, and the last instruction in each of the semantic routines is a branch instruction. The method and system first determines the block length of each of the semantic routines. When a first guest instruction is encountered, the block of instructions in a first semantic routine corresponding to a guest instruction is executed. The method and system further comprises using the block length of first semantic routine to determine when to fetch a second semantic routine without fetching and decoding the branch instruction in the first semantic routine, thereby increasing emulation performance.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram depicting a high-level architecture of a host processor.

FIG. 2 is a block diagram illustrating an emulation assist unit.

FIG. 3 is a block diagram illustrating a semantic buffer.

FIG. 4 is a timing diagram of the processor cycles necessary to execute four semantic routines.

FIG. 5 is a block diagram depicting a guest cache.

FIG. 6 is a timing diagram of the processor cycles necessary to execute four semantic routines when the block length is utilized in accordance with the present invention.

DESCRIPTION OF THE INVENTION

The present invention relates to an improvement in emulating guest instructions in a host processor. The following description is presented to enable one of ordinary skill in the art to make and use the invention and is provided in the context of a patent application and its requirements. Various modifications to the preferred embodiment will be readily apparent to those skilled in the art and the generic principles herein may be applied to other embodiments. Thus, the present invention is not intended to be limited to the embodiment shown but is to be accorded the widest scope consistent with the principles and features described herein.

FIG. 1 is a block diagram depicting a high-level architecture of a host processor **10**. Typically, the processor **10** executes native software instructions using various hardware components, such as a memory **12**, a level 2 (L2) cache **14**, a data cache **16**, an instruction cache (IC) **18**, an instruction queue (IQ) **20**, a branch prediction unit **22**, functional units **24–32**, and a completion buffer **34**.

In a preferred embodiment of the present invention, the processor **10** is a PowerPC™ manufactured by Motorola that executes a PowerPC™ (PPC) instruction set. Besides executing the PPC instruction set, the processor **10** is also capable of emulating a plurality of guest instructions **38**, such as Intel x86 instructions and Motorola 68K Complex Instruction Set Computers (CISC) instructions.

The component responsible for assisting the processor **10** in emulating guest instructions **38** is an emulation assist unit (EAU) **36**. To enable the processor **10** to emulate the guest instructions **38**, each guest instruction **38** is first translated into a corresponding set of host instructions, called a semantic routine **40**, that perform the function of guest instruction **38** in the host processor **10**. Each semantic routine **40** is stored at a specific address location in the memory **12**. The guest instructions **38** (and data) are also stored in the memory **12**.

In operation, the processor **10** caches the guest instructions **38** and data into the L2 cache **14** and the data cache **16**, respectively. The EAU **36** fetches and decodes the guest instructions **38**, and then maps each of the guest instructions **38** into the host memory address of the corresponding semantic routine **40**. The processor **10** fetches the semantic routine **40** located at the specified memory address into the instruction cache **18**. The host instructions contained in the semantic routine **40** are then transferred to the IQ **20**. Each host instruction is then decoded and executed using the branch prediction unit **22** and the functional units **24–32**.

After the semantic routine **40** is executed and the results stored in the completion buffer **34**, the EAU **36** maps the host address of the next guest instruction **38**, and the process repeats.

Referring now to FIG. **2**, a block diagram of the EAU **36** is shown. The EAU **36** includes an instruction buffer **50**, a pre-fetch unit **52**, special purpose registers (SPRs) **54**, a decode unit **56**, a branch history table **58**, a tag generation unit **60**, a guest cache (gcache) **62**, and a gcache instruction queue **64**. The function of the EAU **36** is assist the processor **10** in emulating guest instructions **38** using a direct mapping scheme.

A guest instruction **38** is first pre-fetched by the pre-fetch unit **52** from the data cache **16**. The guest instruction **38** is then loaded into the instruction buffer **50** and transferred to the decode unit **56**. Since a direct mapping of a guest instruction **38** to a host address is complicated by its syntax, the decode unit **56** is used to parse or decode a guest instruction **38** to determine length, branch type, and immediate data. If the instruction **38** is a branch, the branch history table **58** is used to predict the next pre-fetch address.

The gcache **62** is an array that includes an entry for each semantic routine **40** and its corresponding host memory address. After the guest instruction **38** is decoded, the tag generation unit **60** converts the instruction **38** into a unique opcode tag, and the opcode tag is used as an index to the gcache **62** to access the corresponding semantic routine **40** address.

As guest instructions **38** are decoded and the corresponding semantic routines **40** become known, the semantic routines **40** and their addresses are stored in the gcache instruction queue **64**. Each entry in the instruction queue **64** contains the host address of a semantic routine **40**, the immediate data that was contained in the guest instruction **38**, and a guest instruction pointer.

Referring to both FIGS. **1** and **2**, the processor **10** fetches the first entry in the instruction queue **64** via line **66**, and retrieves the appropriate semantic routine **40** from memory **12** for execution. The last instruction in each semantic routine **40** is a branch instruction, which in a preferred embodiment, is a guest dispatch complete (gdisc) instruction **68**. When the processor **10** fetches and decodes a gdisc instruction **68**, it causes the processor **10** to fetch, via line **68**, the address of the next semantic routine **40** from the instruction queue **64**.

In a preferred embodiment, the address of the next semantic routine **40** to be executed is stored in the current instruction pointer (IP) **70**. The decode of the gdisc instruction **68** also causes the next IP **72** to be moved into the current IP **72**. Also in a preferred embodiment, the gdisc instruction is an unconditional branch instruction, but any instruction that causes the processor to fetch the next semantic routine may be used.

As stated above, the semantic routines **40** are stored in memory **12**. In a preferred embodiment, the semantic routines are stored in a semantic buffer within the memory **12**.

Referring now to FIG. **3**, a block diagram of a semantic buffer **80** is shown. The semantic buffer **80** contains a plurality of semantic routines **40**, shown as semantic routines A–N. Each of the semantic routines **40** include a block **82** of instructions **82**. The last instruction in each of the blocks **82** is an unconditional branch instruction **68**. The number of instructions in each block **82** determine the length of the block **82**. For example, semantic routine A includes three instructions followed by one branch instruction **68**. Therefore, semantic routine A has a block length of four.

Referring now to FIG. **4**, a timing diagram of the processor cycles necessary to execute semantic routines A through D is shown. The processor **10** executes instructions in a pipeline process that includes a fetch cycle, decode cycle, and an execute cycle.

In a preferred embodiment, the processor **10** has a bandwidth of two, meaning that the processor **10** is capable of fetching two instructions at once during a fetch cycle.

Assume that in FIG. **4**, the processor **10** is to execute semantic routines A through D in sequence. In cycle **1**, instructions A1 and A2 of semantic routine A are fetched. In cycle **2**, instructions A1 and A2 are decoded while instructions A3 and BRA are fetched. In cycle **3**, instructions AI and A2 are executed while instructions A3 and BRA are decoded. The next fetch address is unknown until the BRA instruction is decoded. Therefore, no instructions are fetched during this cycle.

In cycle **4**, only instruction A3 is executed since BRA is only decoded, not executed, and instructions B1 and BRB are fetched. In cycle **5**, instructions B1 and BRB are decoded. Since the branch address for semantic routine C is unknown at this time, instructions C1 and C2 are not fetched until cycle **6**. The instructions for semantic routines C and D are fetched, decoded, and executed in a similar manner. Overall, nine execution cycles (cycles **3–11**) were required to execute semantic routines A–D.

As shown in FIG. **4**, a processor cycle is wasted every time the processor **10** is forced to fetch and decode a branch instruction **68** to determine when to fetch the next semantic routine **40**. The addition of these extra cycles results in a performance penalty.

The present invention is a method and system for minimizing the number of cycles required to execute semantic routines. According to the present invention, the block lengths of each of the semantic routines is utilized to determine when to fetch the next semantic routine, which eliminates the need to fetch and decode the gdisc branch instructions **68** and saves processor cycles. To more particularly illustrate the present invention, refer now to FIG. **5**.

FIG. **5** is a block diagram of a gcache **62**. Each entry in the gcache instruction queue (IQ) **64** typically comprises an offset field **100** and a data field **102**. The offset field **100** is a pointer to the host (PowerPC (PPC)) address of the semantic routines **40**. The data field **100** includes the immediate data from the guest instruction **38** that the semantic routine **40** is to process.

According to the present invention, each entry in the gcache IQ **64** is provided with a length field **104** that contains the block length of the corresponding semantic routine **40**. In a preferred embodiment, the block length is determined when a semantic routine **40** is compiled.

Referring to both FIGS. **2** and **5**, when the processor **10** reads an entry from the gcache IQ **64**, the value in length field **104** is stored within a counter in the processor **10**, and is used to indicate how many instructions will occur before a gdisc branch instruction **68** is encountered. As each instruction in the semantic routine is fetched, the counter containing the value in the length field **104** is decremented. A value of "1" indicates that the next instruction in the semantic routine is a gdisc branch instruction **68**. At this point, the processor **10** uses the current instruction pointer **70** to fetch the target address of the next semantic routine **40** without having to fetch and decode the branch instruction **68**.

The fetch bandwidth of the processor may cause a branch instructions **68** to be fetched even though the value of in

counter indicates otherwise. A branch instruction will be fetched even though the counter is less than or equal to one when the block length of the semantic routine divided by the fetch bandwidth results in an even value. For example, referring again to FIG. 3, semantic routine A includes four instruction. Assuming that the fetch bandwidth is two, then the block length (four) divided by the fetch bandwidth (2) is two, an even number, and the BRA instruction will be fetched. This principle is illustrated in the following example.

FIG. 6 is a timing diagram of the processor cycles necessary to execute semantic routines A through D when the block length is utilized in accordance with the present invention. In cycle 1, instructions A1 and A2 of semantic routine A are fetched. At this point, the value of the length field 104 (not shown) is decremented from four to two.

In cycle 2, instructions A1 and A2 are decoded while instructions A3 and BRA are fetched. Since the processor fetches two instructions at once, BRA was also fetched, and the value of length is decremented from two to zero. If the processor 10 only fetched one instruction at a time then the fetch of BRA would not occur.

In cycle 3, instructions A1 and A2 are executed, and instruction A3 is decoded. Since the value of length is less than one, the BRA instruction is not decoded, and the processor 10 fetches instructions B1 and BRB in the same cycle. Using the length of semantic routine A, the processor 10 was not required to wait until the next cycle to determine the address of semantic routine B, thus saving a processor cycle.

In cycle 4, only instruction A3 is executed, and instruction B1 is decoded. Since the length field for semantic routine B was decremented from 2 to zero and is now is less than one, the BRB instruction is not decoded, and the processor 10 fetches instructions C1 and C2 in the same cycle. The instructions for semantic routines C and D are fetched, decoded, and executed in a similar manner. It can be seen from FIGS. 4 and 6 that the execution time for semantic routines A–D has been reduced from nine to five cycles (cycles 3–7) when using the block length in accordance with the present invention.

As stated above, the counter containing the value of the length field 104 can be decremented to zero when the processor fetches two instructions at once. A zero value in the counter can also occur in other ways. In a preferred embodiment of the present invention, the counter used to store the value of the length field is non-architected. Therefore, when an interrupt occurs within the processor 10, the value of the length field will be destroyed. After an interrupt, the counter is restored to zero. A zero value will also occur when the length of a semantic routine could not be determined because the routine included a conditional branch instruction. In any of the above cases, when the value of length in the counter is zero, the processor fetches the next instruction in the semantic routine 40.

In an alternative embodiment, the gdisc branch instruction is removed from the semantic routines 40 to save memory. In this embodiment, however, the value of the counter needs to be saved, or the number of instructions left to execute in the semantic routine must be calculated, when an interrupt occurs.

A method and system has been disclosed that minimizes the number of cycles required to execute semantic routines through the use of block lengths, thereby increasing emulation performance. Although the present invention has been described in accordance with the embodiments shown, one of ordinary skill in the art will readily recognize that there could be variations to the embodiments and those variations would be within the spirit and scope of the present invention. Accordingly, many modifications may be made by one of ordinary skill in the art without departing from the spirit and scope of the appended claims.

What is claimed is:

1. A method for reducing the cycle time necessary to execute semantic routines in a processor that emulates guest instructions, each of the semantic routines including a block of host instructions, wherein the last instruction in each of the semantic routines is a branch instruction, the method comprising the steps of:

(a) determining a block length for each of the semantic routines;

(b) executing the block of instructions in a first semantic routine corresponding to a first guest instruction; and

(c) using the block length of the first semantic routine to determine when to fetch a second semantic routine without fetching and decoding the branch instruction in the first semantic routine, thereby increasing emulation performance.

2. A method as in claim 1 wherein step (a) further includes the step of:

(a1) storing the block length in a counter.

3. A method as in claim 2 wherein step (b) further includes the step of:

(b1) decrementing the counter when each of the instructions in the semantic routine is executed.

4. A method as in claim 3 wherein step (c) further includes the step of:

(c1) fetching the first instruction of the second semantic routine when the counter is less than or equal to one.

5. A method for reducing the cycle time necessary to execute semantic routines in a processor that emulates guest instructions, each of the semantic routines including a block of host instructions, wherein the last instruction in each of the semantic routines is a branch instruction, the method comprising the steps of:

(a) determining a block length of a first semantic routine;

(b) executing the block of instructions in the first semantic routine by performing a fetch, decode, and execute cycle on each of the instructions in the block of instructions;

(c) decrementing the block length when each of the instructions in the first semantic routine is fetched; and

(d) using the block length of first semantic routine to determine when to fetch a second semantic routine without fetching and decoding the branch instruction in the first semantic routine, thereby increasing emulation performance.

6. A method as in claim 5 wherein step (a) further includes the step of:

(a1) storing the block length in a counter.

7. A method as in claim 6 wherein step (c) further includes the step of:

(c1) decrementing the counter when each of the instructions in the semantic routine is executed.

8. A method as in claim 7 wherein step (c) further includes the step of:

(c1) fetching the first instruction of the second semantic routine when the counter is less than or equal to one.

9. An emulation assist unit for assisting a host processor to emulate a set of guest instructions using semantic routines, each of the semantic routines being stored at

specific addresses in memory and including a block of instructions wherein the last instruction in the block is a branch instruction, each of the semantic routines further having a block length, the emulation assist unit comprising:

a cache for associating each of the semantic routines with a memory address;

decode means for parsing each of the guest instructions to determine instruction type;

tag generation means for generating a tag from the instruction type in order to index the cache;

a queue for storing entries from the cache indexed by the tag, each of the entries in the queue include an offset field, a data field, and a length field, wherein the offset field is a pointer to the memory address of one of the semantic routines, and the length field contains the block length of the corresponding semantic routine; and

the processor including means responsive to a first entry in the queue for executing a first semantic routine located at the memory address indicated by the offset field, and means for using the block length to determine when to execute a second semantic routine from a second entry in the queue without fetching and decoding the branch instruction of the first semantic routine, thereby saving processor cycles.

10. An emulation assist unit as in claim **9** wherein the data field includes immediate data from the corresponding guest instruction.

11. An emulation assist unit as in claim **10** wherein the semantic routines are stored sequentially in a semantic buffer.

* * * * *

EXHIBIT 37

Webster's Third New International Dictionary

UNABRIDGED

A GENUINE MERRIAM-WEBSTER

The name *Webster* alone is no guarantee of excellence. It is used by a number of publishers and may serve mainly to mislead an unwary buyer.

Merriam-Webster™ is the name you should look for when you consider the purchase of dictionaries or other fine reference books. It carries the reputation of a company that has been publishing since 1831 and is your assurance of quality and authority.

Library of Congress Cataloging in Publication Data
Main entry under title:

Webster's third new international dictionary of the English language, unabridged: a Merriam-Webster/editor in chief, Philip Babcock Gove and the Merriam-Webster editorial staff.
p. cm.
ISBN 0-87779-201-1

1. English language—Dictionaries. I. Gove, Philip Babcock, 1902–1972. II. Merriam-Webster, Inc.
PE1625.W36 1993
423–dc20 93-10630
CIP

MADE IN THE UNITED STATES OF AMERICA

495051 QP/H009998

perform a gouging operation ⟨carve, ~, shape and grind on this versatile machine —*advt*⟩ ~ *vt* **1 a** *archaic* : to dig up with the snout ⟨~*ing* up the moss . . . in search of acorns —Peter Beckford⟩ **b** : to gouge out or make a furrow in; as **a** : to scoop out or cut away (as blank parts) from a printing surface (as an engraving or electrotype) with a router **b** : to remove (as metal or wood) with a gouge or other hand-operated cutting tool **2 a** : to expel by force : EJECT — usu. used with *out* ⟨whole families are . . . ~*ed out* of house and home —Arthur Murphy⟩ **b** : to cause to emerge esp. from bed : drag out : ROUST ⟨~*ed* . . . from his garret by loud rings at the bell —Floyd Dell⟩ ⟨~*ed* me out of bed to help place the target —A.C.Fisher⟩ ⟨~ the enemies of Calvinism from the utmost keep of their stronghold —V.L.Parrington⟩ **3** : to dig out : come up with : UNCOVER ⟨went . . . to his cellar and ~*ed* out a bottle of port —John Masefield⟩

²rout \"\ *n* -s : an act, process, or result of routing ⟨this house, with its strange clutter . . . gives the effect of ~ —Howard Griffin⟩

³rout \"\ *n* -s [prob. of Scand origin; akin to ON *hrota* barnacle goose] *chiefly dial* : BRANT

⁴rout \"\ *n* -s [MF *route* troop, band, defeat — more at ²ROUT] **1** : a state of wild confusion or disorderly retreat ⟨charging tanks put the infantry to ~⟩ ⟨reason had been clearly put to ~ by nineteenth-century Romanticism —Edmund Wilson⟩ **2 a** : a disastrous defeat : DEBACLE ⟨the battle became a ~, a shambles —*Amer. Guide Series: Texas*⟩ **b** : a precipitate flight ⟨everybody was for saving his own skin in this frantic ~ —L.C.Douglas⟩ **c** : an act or instance of routing ⟨the ~ of the Democrats . . . resulted in the candidacy of Republican incumbents —V.O.Key⟩ ⟨most crushing defeat since its 61–0 ~ last year⟩ **3** *archaic* : a fleeing force ⟨disordered the rank . . . whereupon their men were in ~*s* —Mary Wroth⟩

⁵rout \"\ *vt* -ED/-ING/-S **1 a** : to disorganize completely : put to precipitate flight : DEMORALIZE, STAMPEDE ⟨the large and well-mechanized army . . . had been ~*ed* and was in part surrounded —Upton Sinclair⟩ ⟨charged the main body of Russian cavalry . . . and ~*ed* it —Al Newman⟩ **b** : to defeat decisively : OVERWHELM ⟨suffered the discomfiture of seeing their party ~*ed* at the polls —A.N.Holcombe⟩ ⟨the team ~*ed* their traditional Thanksgiving Day rivals 41–0⟩ **2** : to drive out : cause to disappear : DISPEL ⟨virtues are discredited and decency is ~*ed* —Frank Mac Shane⟩ **syn** see CONQUER

¹route \'rüt, 'raůt, *usu* \|d·+V\ *also* **rout** \'raůt\ *n* -s [ME *rute*, *route*, fr. OF *route* troop, band, route, fr. (assumed) VL *rupta* (*via*), lit., broken way, beaten way, fr. L *rupta*, fem. of *ruptus*, past part. of *rumpere* to break — more at REAVE] **1 a** : a traveled way : ROAD, HIGHWAY ⟨expressways, toll roads, turnpikes, and similar large-scale ~*s* —P.F.Griffin⟩ ⟨because of its position on a water ~, the village soon became a river-traffic center —*Amer. Guide Series: Mich.*⟩ **b** : a means of access : CHANNEL, PATH ⟨preparedness was offered as a ~ to peace —F.L.Paxson⟩ ⟨liberal arts courses . . . as a ~ to the graduate schools —*Univ. of Chicago Round Table*⟩ **2 a** : a regular routine : customary progression **b** : a method of transmitting a disease or of administering a remedy ⟨the air-borne ~ of . . . infection —M.L.Furcolow⟩ ⟨may be injected into . . . patients who cannot for any reason take the material by the oral ~ —*Collier's Yr. Bk.*⟩ **3** : a line or direction of travel : COURSE, TRACK ⟨bayous change their ~*s* with each flood —*Lamp*⟩ ⟨U.S. 19 follows the general ~ of the old Catawba Trail —*Amer. Guide Series: N.C.*⟩ **4 a** : an established itinerary : a selected or regularly traversed passage esp. between two distant points ⟨soon I'll be over the panhandle . . . if I've not drifted north of ~ —C.A.Lindbergh b.1902⟩ ⟨permission . . . for 53 new domestic air ~*s* —*Americana Annual*⟩ ⟨the preponderance of shipping traffic along the north Atlantic ~ —R.S.Thoman⟩; *specif* : a telephone line ⟨a new procedure . . . enables operators to use the circuits on busy ~*s* more efficiently —C.F.Craig⟩ **b** : an assigned territory to be systematically covered ⟨postal ~⟩ ⟨paper ~⟩ **c** : a prescribed manner of shipment that may include selected carriers, junctions, and delivery point ⟨inform me by what ~ they had sent an order —Georgina Grahame⟩ **5** *archaic* : marching orders ⟨our ~ came for a march —Robert Bage⟩ **6** : a horse race of a mile or more ⟨a horse trained for ~*s* does not do well in sprints⟩ **syn** see WAY

²route \"\ *also* **rout** \"\ *vt* -ED/-ING/-S **1 a** : to plan an itinerary for : send by a selected route : DIRECT ⟨~ lines through the richest inland spots —*Amer. Guide Series: Minn.*⟩ ⟨*routed* volunteers to the guerilla frontiers —E.P.Snow⟩ **b** : to divert in a specified direction ⟨~ the high voltage to the various engine cylinders in the correct sequence —*Aircraft Power Plants*⟩ ⟨took to *routing* their business through his . . . colleague —S.H.Adams⟩ **c** : to select the course to be followed by a shipment by designating carrier, intermediate points, junctions, and final delivery **2** : to put (the mail for a postal route) in order for delivery **3** : to prearrange and direct the order and execution of (a series of processes or transactions) in a factory or business : dispatch documents or materials to appropriate destinations ⟨~ an invoice to the accounting department⟩ **syn** see SEND

route agent *n* [¹*route*] : a postal employee accompanying mail being transported by train and receiving, canceling, and delivering mail along the route

route chart *n* : FLOW CHART

route locking *n* : electric locking that prevents the movement of any switch, movable point frog, or derail in advance of a train after it has passed a signal to proceed

route·man \'-mən\ *n*, *pl* **routemen** [¹*route* + *man*] **1 a** : a salesman or deliveryman on an assigned route **b** : the supervisor of a group of news carriers responsible for helping to establish routes, taking care of complaints, making out bills, and paying carriers **2** : a shipyard worker who plans the most efficient routing of work through a department

route march *n* : a practice march in peacetime or one at a distance from the enemy in wartime in which troops maintain the prescribed interval and distance but are not required to keep step, maintain silence, or hold their arms in any one position

¹rout·er \'raůd·ə(r), -aůtə-\ *n* -s [⁵*rout* + -*er*] : one that routs: as **a** : ROUTER PLANE **b** : a machine with a rapidly revolving vertical spindle and cutter for milling out the surface of wood or metal (as in woodworking or photoengraving) **c** or **router-bit** \'-·,-\ : the lip on a bit (as a center bit) that cuts the radius of the nicker **d** : an aircraft worker who shapes sheet metal blanks with a routing machine **e** : an operator of a machine for cutting designs in wooden stock

²rout·er \'rüd·ə(r), 'raůl, |tə(r)\ *n* -s [²*route* + -*er*] : one that routes: as **a** : a clerk who sorts articles according to delivery routes **b** : a floor boy who keeps shoe workers supplied with material

³router \"\ *n* -s [¹*route* + -*er*] : a horse trained for distance races — compare SPRINTER

router plane *n* [¹*router*] : a plane consisting of a horizontal bar with a handle at each end supporting a vertically inserted narrow cutter operated by pulling and pushing and used for cutting recesses and smoothing the bottom of grooves — see PLANE illustration

route step *n* [¹*route*] : the out-of-step manner of executing a route march — used as a military command ⟨we were never given *route step* until after we'd cleared the last of the company streets —Richard Yates⟩

route transposition *n* : encipherment in which the plaintext letters are placed along a more or less complex path (as a set of lines, a zigzag, a spiral) and then copied out in order as they are found to lie along a differently defined path — compare COLUMNAR TRANSPOSITION

routeway \'-·,-\ *n* : ROUTE 3 ⟨the one natural ~ across the desert waste would appear to be the River Nile —L.D.Stamp⟩

¹routh \'rüth, 'raůth\ *n* -s [origin unknown] *chiefly Scot* : PLENTY, ABUNDANCE

²routh \"\ *also* **routhy** \-thi\ *adj* **1** *chiefly Scot* : PLENTIFUL, ABUNDANT **2** *chiefly Scot* : ABOUNDING

¹rou·tine \(')rü'tēn\ *n* -s [F, fr. MF, fr. *route* traveled way] **1 a** : a standard practice : regular course of procedure ⟨the old speakeasy ~ of having a guard inspect you with one eye through a peephole —Robert Shaplen⟩ ⟨the usual ~ of appointment: tutor, 1871–74; assistant professor, 1875–80; professor, 1880–1916 —J.M.Berdan⟩ **b** : the habitual method of performance of established procedures ⟨the matter-of-fact laconic ~ of the hospital —Leslie Rees⟩ ⟨housewives, their ~ quickened by the pace of wartime living —*Monsanto Mag.*⟩ **c** : a repetitive speech or formula ⟨that old ~ about his welfare, and what was best for it —Gregor Felsen⟩ ⟨the "visit your dentist twice a year" ~ —*Spokane (Wash.) Spokesman Rev.*⟩ **2 a** : adherence to a pattern of behavior characterized by mechanical repetition ⟨most of us are blind — made so by custom and ~ —C.S.Kilby⟩ **b** : the quality or state of being humdrum ⟨sell out quickly . . . to avoid the dull ~ of development —*Amer. Guide Series: Nev.*⟩ **3 a** : an established sequence of operations (as in a factory or business establishment) **b** : a sequence of computer instructions for performing a particular task **4 a** : a standardized piece of entertainment or showmanship : ACT, BIT ⟨went through the hat ~ for distinguished guests —*Time*⟩; *specif* : a theatrical number ⟨a breathtaking ~ on a tightrope —*New Yorker*⟩ ⟨sitting out front at rehearsals, he jots down . . . a description of each ~ —H.W.Wind⟩ **b** : a fixed series of dance steps or rhythmic movements ⟨can pick up a difficult tap ~ at a single rehearsal —Agnes de Mille⟩

²routine \(')·¦·\ *adj* **1** : of a commonplace or repetitious character : ORDINARY, USUAL ⟨the level of artistry . . . was altogether ~ and uninspired —Winthrop Sargeant⟩ ⟨read about the spectacular flights, but . . . seldom read anything about the hundreds of ~ flights made over the same country —Harold Griffin⟩ ⟨a few phrases of ~ patriotism —James Joll⟩ ⟨the car performs well with only ~ maintenance⟩ **2** : of, relating to, or in accordance with established procedure ⟨the ~ settlement of boundary disputes —S.F.Bemis⟩ ⟨the ~ use of the blood-pressure test —F.A.Faught⟩ ⟨the purser's ~ book . . . lists everything that has to be done —*Saturday Rev.*⟩ — **rou·tine·ly** *adv* — **rou·tine·ness** *n* -ES

³routine \"\ *vt* -ED/-ING/-S : ROUTINIZE ⟨a horse should not be *routined* in any one test —*Notes on Dressage*⟩ ⟨~*s* the sequence of dialogue —Maurice Zolotow⟩

rou·ti·neer \,rüt°ni(ə)r\ *n* -s [¹*routine* + -*eer*] : one that adheres to or insists on routine ⟨hacks and ~*s* were making the fascinating game of business and finance appear endlessly dull —M.S.Rukeyser⟩

routine orders *n pl* : orders relating to military matters other than operations in the field — compare COMBAT ORDERS

¹routing *n* -s [fr. gerund of ⁵*rout*] **1** : the removal of excess material (as from a printing plate) by cutting, milling, or gouging **2** : a groove or indentation produced by routing

²routing *n* -s [fr. gerund of ²*route*] **1 a** : a course of travel : ITINERARY **b** : transmission over a selected course : CIRCULATION ⟨selective ~ of messages —Eunice Cooper & Helen Dinerman⟩ **2** : the sorting of mail into proper sequence for delivery **3** : the scheduling or standardization of a flow of work **4 a** : the determination of a course and method of shipment (as of freight) **b** : ROUTE 4c

rou·tin·ist \rü'tēnəst\ *n* -s [¹*routine* + -*ist*] : ROUTINEER

rou·tin·iza·tion \,rü,tēnə'zāshən\ *n* -s : an act or instance of routinizing ⟨specialization and ~ have reduced the creative aspects of their work —R.K.Burns⟩

rou·tin·ize \rü'tē,nīz\ *vt* -ED/-ING/-S [¹*routine* + -*ize*] : to discipline in or reduce to a routine ⟨*routinized* assembly-line workers —J.B.Martin⟩ ⟨we can ~ this process —F.J.Gruenberger⟩

rout·ous \'raůd·əs\ *adj* [²*rout* + -*ous*] *archaic* : NOISY, UPROARIOUS

rout·ous·ly *adv* [*routous* + -*ly*] **1** *archaic* : UPROARIOUSLY, NOISILY **2** : in violation of a law against routs

routs *pl of* ROUT, *pres 3d sing of* ROUT

rout-seat \'·,·\ *n* [²*rout*] *Brit* : a light bench supplied for parties ⟨knocked . . . off the end of a *rout-seat* at a ball —W.F. DeMorgan⟩

roux \'rü\ *n*, *pl* **roux** \'rü(z)\ [F, fr. (*beurre*) *roux* browned butter, fr. *roux* reddish brown, russet, fr. OF *rous* — more at RUSSET] : a mixture of flour and fat cooked sometimes until the flour browns and used to thicken soups and sauces

¹rove [ME *roof*] *past of* RIVE

²rove \'rōv\ *n* -s [ME *rewe*, *rufe*, *rove*, fr. ON *rō*] **1** *or* **roove** \'rüv\ : BURR 3 b **2** : the bight of a rope sling that receives the hook

³rove \"\ *vb* -ED/-ING/-S [ME *roven*] *vi* **1** : to shoot at rovers in archery **2 a** : to move aimlessly : ROAM, STRAY ⟨criminals . . . *roving* about freely without either arrest or custodial restraint —H.E.Barnes⟩ ⟨members . . . *roved* restlessly from one committee meeting to another —Allan Nevins⟩ **b** : to follow a random course : RAMBLE, WANDER ⟨at first he did not follow her, his thoughts had *roved* so far —Ellen Glasgow⟩ ⟨feebly his glance *roved* over the figures by the bed —Mary Austin⟩ **3** *obs* **a** : to deviate from the point ⟨from that mark how far they ~ —John Milton⟩ **b** : to take random aim **c** : GUESS **4** *archaic* : to troll with live bait ⟨~ for a perch with a minnow —Izaak Walton⟩ **5** *dial Brit* : to be light-headed or delirious : RAVE ~ *vt* : to traverse aimlessly : wander through or over : ROAM ⟨permit their progeny . . . to ~ the forest —S.H.Adams⟩ ⟨letting her eyes ~ the room as if she were planning . . . its decoration —Jean Stafford⟩ ⟨saw the searchlights *roving* the sky —Howard Hunt⟩

⁴rove \"\ *n* -s : an act or instance of wandering ⟨a sidelong ~ of the eye —A.L.Kroeber⟩

⁵rove *n* -s [modif. of Sp & Pg *arroba*] *obs* : ARROBA

⁶rove *past of* REEVE

⁷rove \'rōv\ *vt* -ED/-ING/-S [origin unknown] : to join (textile fibers) with a slight twist and draw out into roving

⁸rove \"\ *n* -s [origin unknown] : ³ROVING 1

rove beetle \'rōv-\ *n* [perh. fr. ³*rove*] : any of numerous beetles constituting the family Staphylinidae, having a long body and very short wing covers beneath which the wings are folded transversely, often occurring on decaying animal and vegetable matter, and many of them being predatory and the larvae of some parasitic

rove-over \'·,··\ *adj* [prob. fr. ³*rove* + *over*] : having an extrametrical syllable at the end of one line that forms a foot with the first syllable of the next line — used of a type of verse in sprung rhythm

¹ro·ver \'rōvə(r)\ *n* -s [ME, fr. MD *rover* robber, plunderer, fr. *roven* to rob + -*er*; akin to OE *rēafian* to reave — more at REAVE] : PIRATE ⟨the accumulated loot of all the sea ~*s* —H.E.Rieseberg⟩

²rov·er \"\ *n* -s [ME, fr. *roven* to shoot at random, wander + -*er* — more at ROVE] **1 a** : a random mark at an uncertain distance used as a target in archery — usu. used in pl. ⟨in shooting at ~*s* the archer whose arrow comes nearest the mark selects the next target⟩ **b** : one of a series of fixed marks at long range **c** : a strong arrow used in shooting at rovers **d** : an archer shooting at rovers **2** : one that wanders: as **a** : a habitual roamer : TRAVELER, STRAY ⟨as much night life . . . as any vacation ~ can safely stand —C.L.Biemiller⟩ ⟨cattle, some ~*s* always excepted . . . remain on a given range —J.F.Dobie⟩ **b** : FLIRT, MASHER ⟨my true love's a ~ —Edna S. V. Millay⟩ **c** *chiefly Brit* : a boy scout over 17 years old who takes part in advanced scouting activities **d** : an architectural molding that follows a curve **e** *usu cap* : COLORADAN — used as a nickname **3 a** *or* **rover ball** : a croquet ball that has been through all the wickets and would be out if it hit the stake but is continued in play **b** : the player of a rover ball — **at rovers** *adv*, *obs* : at random : HAPHAZARDLY ⟨speaketh not *at rovers* —Arthur Golding⟩

³rov·er \"\ *n* -s [⁷*rove* + -*er*] : one that makes roving

rover bellflower *n* [²*rover*] : a coarse European perennial herb (*Campanula rapunculoides*) widely naturalized esp. in eastern No. America and having racemes of nodding flowers with campanulate corollas

rov·er boy *n*, *usu cap R&B* [fr. the *Rover boys*, heroes of a series of juvenile books (1899–1925) by Edward Stratemeyer †1930 Am. writer] : a physically brave and morally excellent person of somewhat limited outlook and experience ⟨trying to be the *Rover Boy* of the beachhead —R.M.Ingersoll⟩

rovescio *adv* [by shortening] : a rovescio

¹roving *n* -s [ME, fr. gerund of *roven* to shoot at random, wander — more at ROVE] **1** : an act or instance of shooting at random archery targets and esp. at natural targets in fields or woodlands **2** : an act or instance of roaming ⟨every year this animal's ~*s* are restricted —Washington Irving⟩ ⟨found dial ~ common in the early parts of a program —George Fisk⟩

²roving *adj* [fr. pres. part. of ³*rove*] **1** *obs* : based on guesswork : CONJECTURAL **2 a** : traversing a random course : NOMADIC, WANDERING ⟨a ~ band of gypsies⟩ ⟨a ~ vixen wanting cubs —John Masefield⟩ **b** : traversing an assigned route or capable of being shifted from place to place : MOBILE ⟨~ judge⟩ ⟨~ reporter⟩ ⟨serving as a ~ police force for . . . the Territory of Arizona —Ross Santee⟩ **c** : of a general nature : unrestricted as to location or area of concern ⟨~ envoy⟩ ⟨~ assignment⟩ **d** : DISCURSIVE, RAMBLING ⟨a ~ wit⟩ ⟨unrelated subjects that happen to strike a ~ fancy —Dorothy Sayers⟩ **3** : inclined to travel or stray : PERIPATETIC, ROAMING ⟨a large and ~ cast subsidiary to the main characters —Sylvia Berkman⟩ ⟨alas for poor Madame, he had a ~ eye —HenryS.Jones⟩

³roving *n* -s [fr. gerund of ⁷*rove*] **1 a** : a slightly twisted roll or strand of textile fibers **b** : material in an intermediate stage between sliver and yarn **2** : the final process of reducing and drawing out sliver preliminary to spinning — compare SLUBBING

roving reel *n* [³*roving*] : a device for measuring the length of textile rovings

¹row \'rō\ *vb* -ED/-ING/-S [ME *rowen*, fr. OE *rōwan*; akin to MHG *rüejen* to row, ON *rōa*, L *remus* oar, Gk *eressein* to row, *eretmon* oar, Skt *aritra*] *vi* **1 a** : to propel a boat by means of oars ⟨got into the dinghy and ~*ed* out to the sloop⟩ **b** : to be a member of a racing crew ⟨~*ed* on the varsity eight⟩ **c** : to take part in a rowing competition ⟨~*s* against the champions in the annual regatta⟩ **2** *archaic* : to struggle to advance ⟨no one shall find me ~*ing* against the stream . . . I write for general amusement —Sir Walter Scott⟩ **3** : to move by or as if by the propulsion of oars ⟨as the boats ~*ed* in . . . we could hear groans and lamentations —Kenneth Roberts⟩ ⟨pelicans ~ by on slow, powerful wings —Juana Vogt⟩ ~ *vt* **1 a** : to propel with or as if with oars ⟨~ a boat⟩ **b** : to be equipped with (a specified number of oars) ⟨the ceremonial barge ~*ed* 14 oars⟩ **c** (1) : to participate in (a rowing match) ⟨~ a race⟩ (2) : to compete against in a rowing match ⟨~*s* the champion in the regatta⟩ (3) : to pull (an oar) in a crew ⟨~*ed* stroke for the class crew⟩ **2** : to transport in or as if in a boat propelled by oars ⟨charged a small fee to ~ us across the river⟩ ⟨sailors on shore leave ~ their girls around the lake in the park⟩

²row \"\ *n* -s : an act or instance of rowing ⟨go for a ~ on the lake⟩

³row \'raů\ *chiefly Scot var of* RAW

⁴row \'rō\ *n* -s [ME *rawe*, *rowe*, fr. OE *rāw*, *ræw*; akin to OHG *rīga* line, L *rima* slit, fissure, crack, Skt *rikhati* he scratches, *rekhā* scratch, line] **1 a** : a number of objects in an orderly series : STRING ⟨a double ~ of sodium vapor highway lamps —*Amer. Guide Series: Va.*⟩ **b** : an uninterrupted sequence : SUCCESSION ⟨utter . . . ~*s* of platitudes —Joyce Cary⟩ ⟨won the state tourney for four years in a ~ —*Bull. of Bates Coll.*⟩ **c** : an arbitrary series or arrangement of the twelve-tone chromatic scale used as a basis or organizational device for modern musical compositions **2** *archaic* : a homogeneous group : CATEGORY, SET ⟨an only daughter . . . who is, at least, approaching the old maid's ~ —Manasseh Cutler⟩ **3** *obs* : a written line esp. metrical ⟨the first ~ of the pious chanson —Shak.⟩ **4 a** : ¹BLOCK 5c(2) ⟨street after street exactly alike, lined with ~*s* —T.F.Hamlin⟩ **b** : a way for passage : ALLEY, STREET ⟨on Catfish ~ and down Ramcat Alley —Shelby Foote⟩ ⟨two of the island's main arteries, Royal Poinciana Way and Coconut ~ —Walter Cartwright⟩ **c** : a street or area dominated by a specific kind of enterprise or occupancy ⟨in most cities a separate automobile ~ has arisen on the edge of the central business district —C.D.Harris & E.L. Ullman⟩ ⟨rumors fly along diplomatic ~⟩ ⟨zigzag from movie house to movie house like a barfly on whiskey ~ —Nathaniel Bart⟩ **5** : a continuous strip usu. running horizontally or parallel to a base line: as **a** : a line of seats in a theater ⟨a pair of seats in the fifth ~ center⟩ **b** : a line of cultivated plants ⟨hoe between the ~*s*⟩ **c** : a horizontal line (as of figures) — distinguished from *column* ⟨~ totals are added to get the column total⟩ **d** : a line of stitches across a piece of needlework ⟨a ~ of knitting⟩ **e** (1) : a line of tufts in a carpet ⟨there is usu. one ~ of pile tufts for each cycle of back weaving⟩ (2) : the average number of tufts per inch in a carpet counted in the direction of the warp — **a row to hoe** : a task to accomplish — usu. used with a modifying adjective ⟨have had *a* hard *row to hoe* —*Coast Artillery Jour.*⟩ ⟨I'd like to be a surgeon, but it's *a* long *row to hoe* —*Sat. Eve. Post*⟩

⁵row \"\ *vt* -ED/-ING/-S : to form into or furnish with rows ⟨above the . . . heads of the students ~*ed* before me —Ralph Ellison⟩ ⟨a bare room ~*ed* with dusty windows —R.M. Coates⟩

⁶row \'raů\ *n* -s [origin unknown] **1 a** : a noisy disturbance : BRAWL, RUCKUS ⟨a first-class ~ between a brutal ranger . . . and an inoffensive citizen —S.E.White⟩ **b** : a heated argument : QUARREL, SQUABBLE ⟨a terrific ~ . . . between husband and wife because the former put a 15¢ stamp too much on a letter —H.J.Laski⟩ ⟨during the recent ~ over atomic-energy legislation their feuding was epic —Alfred Friendly⟩ **2** *slang chiefly Brit* **a** : a loud sound : NOISE, RACKET ⟨would make a beastly ~ with that instrument —F.M. Ford⟩ **b** : MOUTH ⟨she give him a big apple to shut his ~ —Richard Llewellyn⟩ **syn** see BRAWL

⁷row \"\ *vb* -ED/-ING/-S *vt* **1** *archaic* : to subject to assault : rough up **2** *chiefly Brit* : to speak angrily to : BERATE, SCOLD ⟨~*ed* the driver about the fare —*McClure's*⟩ ~ *vi* : to have a quarrel : FIGHT, SQUABBLE ⟨wrangled and ~*ed* with . . . other editors —W.A.White⟩

ROW *abbr* right of way

row·able \'rōəbəl\ *adj* [¹*row* + -*able*] : capable of being rowed or rowed upon

row·an \'raůən, 'rōən\ *n* -s [of Scand origin; akin to ON *reynir* rowan; akin to OE *rēad* red — more at RED] **1** : ROWAN TREE **2** : ROWANBERRY

row·an·berry \'·-·- — *see* BERRY\ *n* [*rowan* + *berry*] : the fruit of the rowan tree

rowan tree *n* **1** : a Eurasian tree (*Sorbus aucuparia*) with pinnate leaves and flat corymbs of small white flowers followed by red pomes resembling berries — called also *European mountain ash* **2** : AMERICAN MOUNTAIN ASH

row-barge \'rō,·\ *n* [¹*row* + *barge*] : a ship's barge or a passenger boat propelled by oars

row binder \'rō,··\ *n* [⁴*row*] : CORN BINDER

rowboat \'rō,·\ *n* [¹*row* + *boat*] : a small boat of shallow draft usu. having a flat or rounded bottom, a squared-off or V-shaped stern, cross thwarts for rowers and passengers, and rowlocks for the oars with which it is propelled

row crop \'rō-\ *n* [⁴*row*] : a crop (as corn or cotton) that is usu. planted in rows

row culture \'rō-\ *n* : cultivation of crops in rows

row·di·ly \'raůd°l|ē, -dǝl|, |i\ *adv* : in a rowdy manner

row·di·ness \-dēnǝs, -din-\ *n* -ES : the quality or state of being rowdy

row down \(')rō-\ *vt* [¹*row*] : to overtake (as another racing shell) in a rowing match

¹row·dy \'raůdē, -di\ *adj* -ER/-EST [perh. irreg. fr. ⁶*row*] **1 a** : lacking in refinement : noisily turbulent : BOISTEROUS, ROUGH ⟨a shouting ~ game —Marjory S. Douglas⟩ ⟨a ~ little girl, who gave way upon the slightest provocation to uncontrollable laughter —Scott Fitzgerald⟩ ⟨~ mountain brooks —*Amer. Guide Series: Conn.*⟩ **b** : of a disreputable character : RAFFISH, VULGAR ⟨lackeys, housemaids and yokels of all sorts formed the most ~, but also the most enraptured, group in the . . . theater —W.S.Clark⟩ ⟨the comedy is often broad, even ~ —Hollis Alpert⟩ **2** *Austral* : lacking in docility : STUBBORN, UNRULY — used of livestock

²rowdy \"\ *n* -ES : one that is boisterous or pugnacious ⟨a favorite pastime of these . . . *rowdies* was to ride through the town at great speed while shooting with both hands —S.H. Holbrook⟩ ⟨the tough eggs, the *rowdies* in the crew —E.L. Burdick⟩

³rowdy \"\ *vi* -ED/-ING/-ES : to behave in a rowdy manner ⟨there was a lot of gambling and *rowdying* —Bruce Siberts⟩

row·dy·dow *or* **row-de-dow** \'raůdē,daů\ *n* -S [irreg. fr. ⁶*row*] **1** : noisy excitement : HUBBUB, TO-DO ⟨piqued by all the ruffle and ~ —M.G.Bishop⟩ **2 a** : a noisy disturbance or spirited contest : BRAWL, FIGHT **b** : a boisterous party : SPREE

row·dy·dow·dy \¦raůdē¦daůdē\ *adj* [irreg. fr. *rowdydow*] : BOISTEROUS, VULGAR

EXHIBIT 38

MICROSOFT PRESS®

COMPUTER DICTIONARY

SECOND EDITION

COMPLETELY REVISED AND UPDATED, WITH NEW DEFINITIONS AND ILLUSTRATIONS

THE COMPREHENSIVE STANDARD FOR BUSINESS, SCHOOL, LIBRARY, AND HOME

Microsoft PRESS

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data

Microsoft Press computer dictionary : the comprehensive standard for
business, school, library, and home / Microsoft Press. -- 2nd ed.
p. cm.
ISBN 1-55615-597-2
1. Computers--Dictionaries. 2. Microcomputers--Dictionaries.
I. Microsoft Press. II. Title: Computer dictionary.
QA76.15.M54 1993
004'.03--dc20 93-29868
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 MLML 9 8 7 6 5 4

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging-in-Publication Data available.

Project Editor: Casey D. Doyle
Manuscript Editor: Alice Copp Smith
Technical Editors: Mary DeJong, Jeff Carey, Dail Magee, Jr., Jim Fuchs, Seth McEvoy

to the newly vacated position at the opposite end of the register. *Compare* shift.

rotational delay Also called rotational latency. The time required for a desired disk sector to rotate to the read/write head.

RO terminal Abbreviation for read-only terminal. A terminal that can receive data but cannot create data. Nearly all printers can be classified as RO terminals.

round In mathematics, to shorten the fractional part of a number, usually increasing or decreasing the last remaining (rightmost) digit according to whether the deleted portion was over or under 5; for example, 0.3333 rounded to two decimal places is 0.33, and 0.6666 is 0.67. Computer programs often round numbers, sometimes causing confusion when the resulting values do not add up "correctly." Percentages in a spreadsheet can thus total 99 percent or 101 percent because of rounding.

round robin A sequential, cyclical allocation of resources to more than one process or device.

router An intermediary device on a communications network that expedites message delivery. On a single network linking many computers through a mesh of possible connections, a router receives transmitted messages and forwards them to their correct destinations over the most efficient available route. On an interconnected set of local area networks (LANs) using the same communications protocols, a router serves the somewhat different function of acting as a link between LANs, enabling messages to be sent from one to another. *See also* gateway.

routine A generic term for any section of code that can be invoked (executed) within a program. A routine usually has a name (identifier) associated with it and is executed by referencing that name. Related terms, which might or might not be exact synonyms, depending on the context, are *function, procedure,* and *subroutine*. *See also* function, procedure, subroutine.

row A series of items arranged horizontally within some type of framework—for example, a continuous series of cells running from left to right in a spreadsheet; a horizontal line of pixels on a video screen; or a set of data (text or numeric values) aligned horizontally in a table, a mathematical array, or some other row-and-column type of matrix. *Compare* column.

RPN Reverse Polish notation. *See* postfix notation.

RPROM Reprogrammable PROM. *See* EPROM.

RS-232-C standard An accepted industry standard for serial communications connections. Adopted by the Electrical Industries Association, this Recommended Standard (RS) defines the specific lines and signal characteristics used by serial communications controllers to standardize the transmission of serial data between devices. The letter *C* denotes that the current version of the standard is the third in a series.

RS-422/423/449 Standards for serial communications with transmission distances over 50 feet. RS-449 incorporates RS-422 and RS-423. Macintosh serial ports are RS-422 ports. *See also* RS-232-C standard.

RTF *See* Rich Text Format.

RTS Abbreviation for Request To Send, a signal used in serial communications; sent, as from a computer to its modem, to request permission to transmit. RTS is a hardware signal sent over line 4 in RS-232-C connections. *Compare* CTS; *see also* RS-232-C standard.

rubber banding In computer graphics, changing the shape of an object made up of connected lines by "grabbing" a point on an anchored line and "pulling" it to the new location. The illustration shows a line before and after rubber banding has taken place.

Rubber banding.
An object before (left) and after (right) rubber banding.

rule A line printed above, below, or to the side of some element, either to set that item off from the remainder of the page or to improve the look of

EXHIBIT 39

MICROSOFT PRESS®

COMPUTER DICTIONARY

SECOND EDITION

COMPLETELY REVISED AND UPDATED, WITH NEW DEFINITIONS AND ILLUSTRATIONS

THE COMPREHENSIVE STANDARD FOR BUSINESS, SCHOOL, LIBRARY, AND HOME

Microsoft PRESS

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data

Microsoft Press computer dictionary : the comprehensive standard for
business, school, library, and home / Microsoft Press. -- 2nd ed.
p. cm.
ISBN 1-55615-597-2
1. Computers--Dictionaries. 2. Microcomputers--Dictionaries.
I. Microsoft Press. II. Title: Computer dictionary.
QA76.15.M54 1993
004'.03--dc20 93-29868
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 MLML 9 8 7 6 5 4

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging-in-Publication Data available.

Project Editor: Casey D. Doyle
Manuscript Editor: Alice Copp Smith
Technical Editors: Mary DeJong, Jeff Carey, Dail Magee, Jr., Jim Fuchs, Seth McEvoy

rations usually require a size specification in addition to name and type, and record declarations must specify the elements of the record. Some languages require declarations for all variables; others require declarations only for certain types (often arrays and records); and still others have no declarations. *See also* array, data type, record, variable.

data definition language Abbreviated DDL. A language, usually a part of a database management system, that is used to define all attributes and properties of a database, especially record layouts, field definitions, key fields (and, sometimes, keying methodology), file locations, and storage strategy.

data description language A language designed specifically for declaring data structures and files, often in a machine- or language-independent fashion. *See also* data definition language.

data dictionary A database containing data about all the databases composing a database system. The content of the data dictionary can best be thought of as "data about the data"—that is, descriptions of all of the other objects (files, programs, and so on) in the system. In particular, a data dictionary stores all the various schemas and file specifications and their locations. A complete data dictionary also includes information about which programs use which data and which users are interested in which reports. The data dictionary is frequently integrated into the system it describes.

data directory *See* catalog, data dictionary.

data element Also called a data item. An individual unit of data. A data element (such as a field) is defined for processing purposes and might have a specific size, type, and range. *See also* data field.

data encryption *See* encryption.

Data Encryption Standard Abbreviated DES. A commonly used, highly sophisticated algorithm developed by the U.S. National Bureau of Standards for encrypting and decrypting data. *See also* encryption.

data entry The process of writing new data into computer memory, typically from a keyboard.

data field A well-defined portion of a data record, such as a column in a database table; also, the physical representation of such a data unit.

data field masking The process of filtering or otherwise selecting a portion of a data field in order to control the manner in which it is returned and (possibly) displayed.

data file A file consisting of data—text, numbers, or graphics—as distinct from a program file of executable instructions. *Compare* program file.

data flow The movement of data through a system, from its entry to its destination. Data flow can be as simple as input-process-print-store. It can also be a complex matter involving one or more programs or communication from one computer to another in a network, including intermediate stations (nodes) along the way. Data flow is a concern of both system and network administrators and of people who analyze the movement of information through a system. In the form of computing known as parallel processing, *dataflow* (one word) refers to a type of design in which a calculation can be made either when all necessary information is available (known as data-driven processing) or when other processors request the results of the calculation (known as demand-driven processing). *See also* parallel processing.

dataflow *See* data flow.

data fork On the Macintosh, the part of a stored document file that typically contains user-supplied information. The data fork differs from the resource fork, which in program files typically contains frequently used information such as blocks of executable program code, dialog boxes, font data, digitized sound, icons, and so on. A file on the Macintosh can have a header (summary information, such as filename and size), a resource fork, and a data fork, all used by the operating system in storing and managing the file. *See also* resource, resource fork.

data format The structure applied to data by an application program. The data format provides the context in which the data can be interpreted by the program (and the user).

data frame A packet of information transmitted

EXHIBIT 40

IBM Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing,* SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

bined into one signal, thus producing a higher-quality picture.

supervisor (1) The part of a control program that coordinates the use of resources and maintains the flow of processing unit operations. See also system supervisor. (2) Synonym for supervisory program, supervisory routine. (3) See overlay supervisor.

supervisor authority In the Folder Application Facility, authorization to add or delete system users, modify user profiles, redirect and suspend document assignments, and monitor all work in the routing queue.

supervisor call (SVC) A request that serves as the interface into operating system functions, such as allocating storage. The SVC protects the operating system from inappropriate user entry. All operating system requests must be handled by SVCs.

supervisor call instruction An instruction that interrupts a program being executed and passes control to the supervisor so that it can perform a specific service indicated by the instruction.

supervisor lock In OS/VS and VM, an indicator used to inhibit entry to disabled code while a disabled page fault is being resolved.

supervisor mode In DPPX, the privilege mode that allows processing of all instructions. See also application mode, I/O mode, master mode.

supervisor-privileged instruction In the IBM 8100 Information System, an instruction that can be executed in supervisor or master modes but not in application or I/O modes.

supervisor services The part of DPCX that provides the environment for one or more DPCX tasks to execute in the 8100 system.

supervisor state A state during which a processing unit can execute input/output and other privileged instructions. Contrast with problem state.

supervisory Pertaining to a frame format that performs data link control functions such as acknowledging information frames, requesting retransmission, and requesting temporary suspension of transmission. Receive ready (RR), receive not ready (RNR), and reject (REJ) are examples of supervisory frame formats.

supervisory program A computer program, usually part of an operating system, that controls execution of other computer programs and regulates the flow of work in a data processing system. (I) (A) Synonymous with executive program, supervisor.

supervisory relay A relay that, during a telephone call, is controlled by the transmitter current supplied to a subscriber line to receive from the associated stations the signals that control the actions of operators or switching mechanisms.

supervisory routine A routine, usually part of an operating system, that controls the execution of other routines and regulates the flow of work in a data processing system. (I) (A) Synonymous with executive routine, supervisor.

supervisory services (1) A general term for all functions in the supervisor that are available to the user. (2) The network control program supervisor code that provides miscellaneous services, such as the communication adapter interface, starting channel output, controlling timer operations, and data manipulation and utility services.

supervisory (S) format A format used to perform data link supervisory control functions, such as acknowledge I frames, request retransmission of I frames, and request temporary suspension of transmission of I frames. See also information format, unnumbered format.

supervisory (S) frame A frame in supervisory format used to transfer supervisory control functions. See also information format, unnumbered format.

supervisory signals Signals used to indicate the various operating states of circuit combinations.

supplemental windows In SAA Advanced Common User Access architecture, windows that enable applications to extend the dialog with a user. The two supplemental windows are secondary windows and dialog boxes. See secondary window, dialog box.

supply reel A spool from which magnetic tape unwinds during operation of a cartridge or cassette. See also take-up reel.

supply voltage indicator On dictation equipment, a device giving an audible or visible indication that the correct voltage is set for a main-powered machine or that the battery is sufficiently active in a battery-powered machine. (I)

support In system development, to provide the necessary resources for the correct operation of a functional unit. (T) See also system support.

suppressible text Text that is to be ignored in certain copy operations.

suppression (1) In AS/400 AFP support, the electronic equivalent of the "spot carbon," which prevents

EXHIBIT 41

IBM

Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing*, SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

data connection The interconnection of two data terminal equipment (DTEs) by means of switched tandem data circuits to enable data transmission to take place between DTEs. (I)

data constant See figurative constant.

data contamination Synonym for data corruption.

data content Synonym for data inventory.

data control block (DCB) A control block used by access method routines in storing and retrieving data.

data conversion The process of changing data from one form of representation to another.

data converter A functional unit that transforms data from one representation to an equivalent representation. (T) (A)

data corruption (1) A violation of data integrity. (T) (2) Synonymous with data contamination.

data count field (DCF) In SNA, a binary count of the number of bytes in the basic information unit (BIU) or BIU segment associated with the transmission header (TH).

data country code (DCC) In X.25 communications, a 3-digit code, unique to each country, that specifies the X.21 call format used by a network in its International Data Number to call another station.

data declaration A nonexecutable statement that describes the characteristics of the data to be operated upon; for example, PICTURE clause, DIMENSION. (T) Synonymous with data declaration statement.

data declaration statement Synonym for data declaration.

data definition (1) A program statement that describes the features of, specifies relationships of, or establishes context of, data. (A) (2) Information that describes the contents and characteristics of a field, record, or file. A data definition can include such things as field names, lengths, locations, and data types. See field definition, file definition; format definition. (3) In the C language, a definition that describes a data object, reserves storage for a data object, and can provide an initial value for a data object. A data definition appears outside a function or at the beginning of a block statement.

data definition (DD) statement A job control statement describing a data set associated with a specific job step.

data definition language (DDL) A language for describing data and their relationships in a database. Synonymous with data description language. (T)

data definition name (ddname) The name of a data definition (DD) statement that corresponds to a data control block that contains the same name.

data density (1) The number of data characters stored per unit of length, area, or volume. (T) (2) The number of bytes per unit of measure on a recording medium; for example, bytes per inch.

data description For data objects that are not self-describing, components of the data object that describe the data so that it may be processed.

data description entry In COBOL, an entry in the Data Division of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

data description language Synonym for data definition language. (T)

data description specifications (DDS) In the AS/400 system and System/38, a description of the user's database or device files that is entered into the system in a fixed form. The description is then used to create files.

data dictionary (1) A database that; for data of a certain set of applications, contains metadata that deal with individual data objects and their various occurrences in data structures. (T) (2) A centralized repository of information about data such as meaning, relationships to other data, origin, usage, and format. It assists management, database administrators, system analysts, and application programmers in planning, controlling, and evaluating the collection, storage, and use of data. (3) In the System/36 interactive data definition utility, a folder that contains field; format, and file definitions. (4) In IDDU, an object for storing field, record format, and file definitions.

data dictionary/directory (DD/D) (1) A database that combines the functions of a data dictionary and a data directory. (A) (2) An inventory of data resources that controls the totality of data elements within an application and that serves as the repository of all descriptive information about each data element, including location information. (A) (3) Loosely, a data dictionary/directory system. (A) (4) Loosely, synonym for information resource dictionary. (A)

data dictionary/directory system (DD/DS) (1) A computer software system that maintains and manages a data dictionary/directory. (A) (2) Loosely,

EXHIBIT 42

IBM

Dictionary of Computing

- The most comprehensive computing dictionary ever published
- More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing,* SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

more

t of a
output

ort or
r field
ending
a doc-
ow to

it least

station
hen a
printed
ation.

nd an
/VS, a
essing
ilities.
ser by
muni-
another
e also
n two
n that
while

e unit
on a
it with
ssion.

system
. alter-
e input
ain of

nmuni-
ead of
nse.

n of a
rnating
system
etween
ractive
essage
essage

conversational monitor system (CMS) A virtual machine operating system that provides general interactive time sharing, problem solving, and program development capabilities, and operates only under control of the VM/370 VM control program.

conversational processing In IMS/VS, an optional facility that allows a user's application program to accumulate information acquired through multiple interchanges with a terminal, even though the program terminates between interchanges.

conversational remote job entry (CRJE) Facility for entering job control language statements from a remote terminal, and causing scheduling and execution of jobs described in the statements. The terminal user is prompted for missing operands or corrections.

conversation group ID An identifier of a particular session between two specific LUs or CPs. Specified in a conversation allocation command, the conversation group ID allows the requested conversation to be allocated to the identified session. This then permits a pair of transaction programs (one in each LU) to serially share use of the designated session with a group of related pairs of transaction programs that use the same conversation group ID.

conversation-level security See session-level security. See also end-user verification.

conversion (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost due to conversion since accuracy of data representation varies among different data types. (I) (2) The process of changing from one method of data processing to another or from one data processing system to another. (3) The process of changing from one form of representation to another; for example, to change from decimal representation to binary representation.

Conversion Reformat Utility A System/38 licensed program that allows a user to run System/3-style sort programs on System/38.

convert (1) To change the representation of data from one form to another, without changing the information they convey; for example, radix conversion, code conversion, analog to digital conversion, media conversion. (T) (2) See also copy, duplicate.

converted command An intermediate form of a character-coded command produced by the VTAM programs through use of an unformatted system services definition table. The format of a converted command is fixed; the unformatted system services definition table must be constructed in such a manner that the character-coded command, as entered by a logical unit, is converted into the predefined converted command format. See also unformatted.

converted journal entry In the AS/400 system and System/38, the version of a journal entry that can be displayed, printed, or written to a database output file.

converter A device that can convert impulses from one form to another, such as analog to digital, parallel to serial, one code to another, or one protocol to another. See code converter, data converter.

converter cache In the AIX operating system, a high-speed buffer storage that contains frequently accessed information associated with a client application. The primary purpose of a converter cache is to reduce access time to key information.

convertible drive In MSS, a drive that can be designated to be either a staging drive or a nonstaging drive.

convex programming In operations research, a particular case of nonlinear programming in which the function to be maximized or minimized and the constraints are appropriately convex or concave functions of the controllable variables. (I) (A) See dynamic programming, integer programming, linear programming, mathematical programming, nonlinear programming, quadratic programming.

conveying roller Synonym for moistening pressure roller.

conveyor delivery mechanism In a duplicator, a delivery system in which the paper is transported into the delivery tray by means of an endless belt system. (T)

coolant distribution unit (CDU) A unit providing liquid cooling to the processor unit. A coolant distribution unit has pumps, a heat exchanger, and its own logic control and power supply.

cooperative application In SAA usage, a type of distributed application, in which the user interface portion of the application runs on a programmable workstation while some or all of the remaining code runs on one or more linked systems.

cooperative processing In SAA usage, the coordinated use of a programmable workstation with one or more linked systems to accomplish a single goal.

coordinate Any of a set of numbers used to specify the location of a point on a line, on a surface, or in space. See absolute coordinate, device coordinate, incremental coordinate, normalized device coordinates, relative coordinate, user coordinate, world coordinate.

EXHIBIT 43

IEEE 100

THE AUTHORITATIVE DICTIONARY OF IEEE STANDARDS TERMS

SEVENTH EDITION

IEEE
Published by
Standards Information Network
IEEE Press

Trademarks and disclaimers

IEEE believes the information in this publication is accurate as of its publication date; such information is subject to change without notice. IEEE is not responsible for any inadvertent errors.

Other tradenames and trademarks in this document are those of their respective owners.

The Institute of Electrical and Electronics Engineering, Inc.
3 Park Avenue, New York, NY, 10016-5997, USA

To order IEEE Press publications, call 1-800-678-IEEE.

Print: ISBN 0-7381-2601-2 *SP1122*

See other standards and standards-related product listings at: http://standards.ieee.org/

Library of Congress Cataloging-in-Publication Data

IEEE 100 : the authoritative dictionary of IEEE standards terms.—7th ed.
p. cm.
ISBN 0-7381-2601-2 (paperback : alk. paper)
1. Electric engineering—Dictionaries. 2. Electronics—Dictionaries. 3. Computer engineering—Dictionaries. 4. Electric engineering—Acronyms. 5. Electronics—Acronyms. 6. Computer engineering—Acronyms. I. Institute of Electrical and Electronics Engineers.

TK9 .I28 2000
621.3'03—dc21 00-050601

Conte

Introdu

How to

Categor

Tradem

The Au

Abstra

Non-IE

The Autl

h card along
ast: vertical
0.10-1994w
:ed in a hor-
:36.10-1996
zing control
ontal deflec-
201-1979w
processor in
rmed. *Con-*

).10-1994w
arized field
) 145-1993
whose elec-

) 145-1993
y polarized
l. *See also:*
/PE) [119]
d vector is
llel to the
olarization
See also:
211-1997
tion is ap-
(PE) [9]
it specifies
ut a given
l microin-

0.12-1990
earchlight)
ie through
rain lies.
TE) [126]
ypewriter,
number of
isplay de-
per of dis-
ical tabu-
).10-1994
ctor char-
e forward
along the
.5-1990w
ng metal,
l or melt-
ietic core
inding.
4-1955w
sectional
te: Nor-
hange of
response
SP) [32]
n which
id that is
45-1993
t having
-1985w
ng of a
shaped
e is one
and two
gaps are
oors on
nission
isform-
open-
ng and
7-1992

horn-gap switch A switch provided with arcing horns. (SWG/PE) C37.36b-1990r, C37.100-1992

horn mouth Normally the opening, at the end of a horn, with larger cross-sectional area. (SP) [32]

horn reflector antenna (1) (communication satellite) A form of reflector antenna, where the energy coming from the throat of a horn is reflected by a segment of a paraboloid. This type of antenna has a very low backlobe. *See also:* Cassegrainian feed. (COM) [24]
(2) An antenna consisting of a portion of a paraboloidal reflector fed with an offset horn that physically intersects the reflector, part of the wall of the horn being removed to form the antenna aperture. *Note:* The horn is usually either pyramidal or conical, with an axis perpendicular to that of the paraboloid. (AP/ANT) 145-1993

horn throat (audio and electroacoustics) Normally the opening, at the end of a horn, with the smaller cross-sectional area. (SP) [32]

horsepower rating, basis for single-phase motor A system of rating for single-phase motors, whereby horsepower values are determined, for various synchronous speeds, from the minimum value of breakdown torque that the motor design will provide. (PE) [9]

hose (1) (liquid cooling) (rotating machinery) The flexible insulated or insulating hydraulic connections applied between the conductors and either a central manifold or coolant passage. (PE) [9]
(2) *See also:* conductor cover. (PE/T&D) 516-1987s

hose clamp *See:* strand restraining clamp.

hoseproof *See:* waterproof machine.

hospital (health care facilities) A building or part thereof used for the medical, psychiatric, obstetrical or surgical care, on a 24-hour basis, of 4 or more inpatients. Hospital, wherever used in this Code, shall include general hospitals, mental hospitals, tuberculosis hospitals, children's hospitals, and any such facilities providing inpatient care. (NESC/NEC) [86]

hospital information system (HIS) An automated system used in hospitals and other health care facilities to perform such tasks as communication between staff members, statistical analysis, inventory planning, and scheduling of medication, blood analysis, and patient testing. *Note:* Hospital information systems typically use interactive operations on a hierarchical file structure based on a patient-oriented record. *Synonyms:* medical information system; health information system. (C) 610.2-1987

Hospital Operating System-Structured Programming Language An application-oriented language with some FORTRAN features, used to manipulate string variables and to support structured programming. (C) 610.13-1993w

host (1) A device to which other devices (peripherals) are connected and that generally controls those devices. (EMC/EMB/MIB) C63.4-1991, 1073.3.1-1994
(2) A device, typically a personal computer, that will control the communications with attached peripherals. (C/MM) 1284-1994
(3) In general, an electronic circuit assembly (e.g., a plug-in unit, a back-plane, a mother board, or another mezzanine card), that provides electrical and/or mechanical connections to a subordinate assembly. (C/BA) 1301.4-1996
(4) Whatever is driving (i.e., providing) commands or data to the printer (e.g., a workstation, a print server, or spooler.) (C/MM) 1284.1-1997
(5) The client or host station/computer, with which the RTU equipment communicates. *Synonym:* master. (PE/SUB) 1379-1997
(6) *See also:* host computer. (C/DIS) 610.7-1995, 1278.2-1995

host byte order The native representation of an integer: unsigned integer m is the representation in *host byte order* of bit string $b_n\, b_{n-1} \ldots b_0$ (where b_n is the *most significant*, or *highest order* bit, and b_0 is the *least significant* or *lowest order* bit) if $m = 2^n * b_n + 2^{n-1} * b_{n-1} + \ldots + 2^0 * b_0$. (C) 1003.5-1999

host character string A sequence of characters describing a host. Within software definition files of an exported catalog, all data that can be encoded using IRV, shall be. Any such data that cannot be so encoded shall be transformed using UTF-8. (C/PA) 1387.2-1995

host computer (1) A computer, attached to a network, providing primary services such as computation, data base access or special programs or programming languages. *See also:* communications computer. (LM/COM/EMB/MIB) 168-1956w, 1073.3.1-1994
(2) A computer that supports one or more simulation applications. All host computers participating in a simulation exercise are connected by network(s) including local area networks, wide area networks, radio frequency links, etc. (DIS/C) 1278.1-1995, 1278.2-1995r
(3) (A) The primary or controlling computer in a multiple computer installation. *Synonyms:* host machine; host. *See also:* bifunctional machine. (B) The primary or controlling processor in a multiprocessor computer or a computer with multiple processing elements, some of which may be dedicated to specific functions. For example, intelligent adapters; math coprocessors. *Synonym:* host processor. (C) 610.7-1995, 610.10-1994
(4) A computer used to prepare computer programs for use on another computer or on another computer system; for example, a computer used to compile, link, or test programs to be used on another system. *Synonym:* host processor. (C) 610.10-1994w

host computer system A computational-based system used for clinical support of patient care, limited to medical device data interchange. *Synonym:* patient care information system. (EMB/MIB) 1073.4.1-2000

hostile environment computer A computer designed for use in an environment not conducive to safe operation, such as one with many dust particles in the air; precautions must be taken to ensure that dust particles do not enter the disk drives or settle on the heads. *See also:* hardened computer. (C) 610.10-1994w

host interface The interface between a communications network and a host computer. (C) 610.7-1995, 610.10-1994w

host language A programming language such as COBOL or PL/I into which data manipulation language statements are embedded. *See also:* data sublanguage. (C) 610.5-1990w

host machine (1) (A) (software) A computer used to develop software intended for another computer. *Contrast:* target machine. **(B) (software)** A computer used to emulate another computer. *Contrast:* target machine. **(C) (software)** The computer on which a program or file is installed. **(D) (software)** In a computer network, a computer that provides processing capabilities to users of the network. (C) 610.12-1990
(2) *See also:* host computer. (C) 610.7-1995

host processor (1) (FASTBUS acquisition and control) The data processing and control processor assigned to exercise overall supervision over a FASTBUS system. Contains detailed knowledge of the system topology. (NID) 960-1993
(2) A processor that controls all or part of a user application network. *See also:* host computer. (C) 610.10-1994w

hot *See:* energized.

hot cathode (thermionic cathode) A cathode that functions primarily by the process of thermionic emission. (ED) 161-1971w

hot-cathode lamp (illuminating engineering) An electric-discharge lamp whose mode of operation is that of an arc discharge. The cathodes may be heated by the discharge or by external means. (EEC/IE) [126]

hot-cathode tube (thermionic tube) An electron tube containing a hot cathode. (ED) 161-1971w, [45]

hot electron programming The injection of energetic "hot" electrons onto the floating gate. (ED) 1005-1998

EXHIBIT 44

IBM

Dictionary of Computing

▼ The most comprehensive computing dictionary ever published

▼ More than 18,000 entries

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing,* SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

each display line. See also vertically displayed records.

horizontal motion index (HMI) On the display screen of an IBM personal computer, a counter below the command area that indicates the position (1 to 80) of the cursor on the writing line.

horizontal pointer In VSAM, a pointer in a sequence-set index record that gives the location of the next sequence-set index record in the sequence set; it is used for keyed sequential access.

horizontal return Movement between the paper carrier and the typing or printing position in a direction opposite to the typing or printing direction, parallel to the typing or printing line.

horizontal tabulation (1) On a printer or a typewriter, movement of the imprint position a predetermined number of character spaces along the writing line. (2) On a display device, movement of the cursor a predetermined number of display positions along a display line. (3) See also vertical tabulation.

horizontal tabulation character (HT) A format effector character that causes the print or display position to move forward to the next of a series of predetermined positions along the same line. (I) (A)

horizontal tabulator key On a typewriter, a control that effects horizontal tabulation. (T)

horizontal wraparound On a display device, the continuation of cursor movement from the last character position in a horizontal row to the first character position in the next row, or from the first position in a row to the last position in the preceding row. Contrast with vertical wraparound.

host (1) In TCP/IP, any system that has at least one Internet address associated with it. A host with multiple network interfaces may have multiple Internet addresses associated with it. (2) In FORTRAN, a program unit that immediately contains an internal procedure. In the case of nested internal procedures, the host is the program unit that immediately contains the internal procedure; for example, if A contains B and B contains C, A is the host of B and B is the host of C, but A is not the host of C.

host access method The access method that controls communication with a domain.

host application program An application program executed in the host computer.

Host Command Facility (HCF) A feature available on a System/370, 43xx, and 30xx host system that enables a user on the host system to use applications on other systems as if they were using remotely attached 5250-type display stations. See also distributed host command facility (DHCF).

host command processor (HCP) The SNA logical unit of the programmable store system store controller.

host computer (1) In a computer network, a computer that usually performs network control functions and provides end users with services such as computation and database access. (T) (2) The primary or controlling computer in a multiple computer installation. (3) A computer used to prepare programs for use on another computer or on another data processing system; for example, a computer used to compile, link edit, or test programs to be used on another system. (4) Synonym for host processor.

host conversational function In PSS, a retail services function that allows an IBM 3653 Point of Sale Terminal or a basic services function for an IBM 3275 Display Station Model 3 to communicate with the host processor.

host ID In TCP/IP, that part of the Internet address that defines the host on the network. The length of the host ID depends on the type of network or network class (A, B, or C). See also Internet address, network ID.

host-initiated program In DPCX, a program executed as a subtask that can be initiated at the request of the host system.

host interface Interface between a network and host computer. (T)

host IOCS In PSS, the input/output control system that handles communication with the host.

host language (1) In a database, a programming language in which a data sublanguage is embedded. (T) (2) In a database management system, pertaining to a programming language, such as COBOL, PL/I, or assembly language, in which the data manipulation capabilities of the database management system are embedded. (A) (3) In SQL, any programming language, such as C, COBOL, FORTRAN, PL/I, and RPG, in which SQL statements can be embedded.

host language system A database management system that is built upon the facilities of a programming language and depends on the application programmer for logical and physical file manipulations. (A)

host logical unit (LU) An SNA logical unit (LU) located in a host processor; for example, an VTAM

EXHIBIT 45

Dictionary of Computing

Limitation of Liability

While the Editor and Publisher of this book have made reasonable efforts to ensure the accuracy and timeliness of the information contained herein, neither the Editor nor the Publisher shall have any liability with respect to loss or damage caused or alleged to be caused by reliance on any information contained herein.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6 5 4 3

ISBN 0-07-031488-8 (HC)
ISBN 0-07-031489-6 (PBK)

The sponsoring editor for this book was Daniel A. Gonneau and the production supervisor was Thomas G. Kowalczyk.

Printed and bound by R. R. Donnelley & Sons Company.

Tenth Edition (August 1993)

This is a major revision of the *IBM Dictionary of Computing,* SC20-1699-8, which is made obsolete by this edition. Changes are made periodically to the information provided herein.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Comments may be addressed to IBM Corporation, Department E37/656, P. O. Box 12195, Research Triangle Park, NC 27709.

International Edition

When ordering this title, use ISBN 0-07-113383-6.

This book is printed on acid-free paper.

device control character A control character used to specify a control function for peripheral devices associated with a system. (T)

device control panel In multimedia applications, an integrated set of controls that is used to manage a device or media object (such as by playing, rewinding, increasing volume, and so on).

device control unit A hardware device that controls the reading, writing, or displaying of data at one or more input/output devices or terminals. See also transmission control unit.

device coordinate (1) In computer graphics, a coordinate specified in a coordinate system that is device dependent. (I) (A) (2) See normalized device coordinates.

device definition In the AIX operating system, information about a device that is in the Customized Database, including attributes and connection locations.

device dependence Reliance on the characteristics of particular types of devices in writing and executing programs or in performing functions. Contrast with device independence.

device-dependent (1) Pertaining to a function that can be accomplished only if particular types of devices are available. (2) Pertaining to a program that can be executed successfully only if particular types of devices are available. (3) Contrast with device-independent.

device description (1) In the OS/400 operating system, an object that contains information describing a particular device or logical unit that is attached to the system. (2) In System/38, an object that contains information describing a particular device that is attached to the system.

device driver (1) A file that contains the code needed to use an attached device. (2) A program that enables a computer to communicate with a specific peripheral device; for example, a printer, a videodisc player, or a CD drive. (3) A collection of subroutines that control the interface between I/O device adapters and the processor.

device emulation The programming that allows one device to appear to the user or to a system as another device. See also display emulation, 3270 emulation, printer emulation.

device-end (DE) In channel operations, a signal from an I/O device that denotes the end of an operation.

device error log A record that contains information about problems that have occurred with devices on the system.

device field (DFLD) In IMS/VS message format service, the smallest area in a device input or output format block whose content and structure are defined by the user.

device file In the AS/400 system and System/38, one of several types of the system object type *FILE. A device file contains a description of how data is to be presented to a program from a device or how data is to be presented to the device from the program.

device font A font particular to a device, such as a video display or printer, and loaded in the memory of the device. Some device fonts have size and language-support restrictions.

device handler In the AIX operating system, the component of a device driver that communicates directly with the hardware. Synonymous with virtual device driver.

device head In the AIX operating system, the component of a device driver that implements the application program interface to a device.

device helper (DevHlp) functions Kernel services (memory, hardware interrupts, software interrupts, queuing, semaphores, and so forth) available to physical device drivers.

device ID A unique identification number assigned to a logical device. A device ID is unique across an entire system.

device independence The capability to write and execute programs or perform functions without regard for the physical characteristics of devices. Contrast with device dependence.

device-independent (1) Pertaining to a function that can be accomplished without regard for the characteristics of particular types of devices. (2) Pertaining to a program that can be executed successfully without regard for the characteristics of particular types of devices. See also symbolic I/O assignment. (3) Contrast with device-dependent. (4) See also symbolic I/O assignment.

device input queue In the network control program, a queue of block control units (BCUs) directed to a device but not yet validated.

device instance In the AIX operating system, a Customized Devices Object Class entry, created when a device is defined. There is a device instance for each device defined in the system.

EXHIBIT 46

IEEE 100

THE AUTHORITATIVE DICTIONARY OF IEEE STANDARDS TERMS

SEVENTH EDITION

IEEE
Published by
Standards Information Network
IEEE Press

The Institute of Electrical and Electronics Engineering, Inc.
3 Park Avenue, New York, NY, 10016-5997, USA

To order IEEE Press publications, call 1-800-678-IEEE.

Print: ISBN 0-7381-2601-2 *SP1122*

See other standards and standards-related product listings at: http://standards.ieee.org/

Library of Congress Cataloging-in-Publication Data

IEEE 100 : the authoritative dictionary of IEEE standards terms.—7th ed.
p. cm.
ISBN 0-7381-2601-2 (paperback : alk. paper)
1. Electric engineering—Dictionaries. 2. Electronics—Dictionaries. 3. Computer engineering—Dictionaries. 4. Electric engineering—Acronyms. 5. Electronics—Acronyms. 6. Computer engineering—Acronyms. I. Institute of Electrical and Electronics Engineers.

TK9 .I28 2000
621.3'03—dc21 00-050601

Conte

Introdu

How to

Categor

Tradem

The Au

Abstra

Non-IE

The Autl

(14) A reference to an integrated circuit or other design structure. (C/TT) 1450-1999

device address The (32-m)-bit identifying number assigned to a FASTBUS device that is compared with the signals on the AD lines during a logical primary address cycle of a FASTBUS operation. The device address is formed by the group and module address fields. The (remaining) low-order m bits are assigned to the internal address field. (NID) 960-1993

device alias A shorthand representation for a device path. (C/BA) 1275-1994

device arguments The component of a node name that is provided to a package's open method to provide addtional device-specific information. (C/BA) 1275-1994

device class-broadcast Selective broadcast-class specified by CSR#7. Controls device response to subsequent cycles within the broadcast. (NID) 960-1993

device communications controller (DCC) A communications interface associated with a medical device. A DCC may support one or more physically distinct devices acting as a single network communications unit. Its purpose is to provide a point-to-point serial communication link to a bedside communications controller (BCC). (EMB/MIB) 1073.4.1-2000, 1073.3.2-2000

device control character (data management) A control character used for the control of auxiliary devices associated with a data processing system or data communication system; for example, a control character for switching such devices on or off. (C) 610.5-1990w

device control language A language used to monitor and/or control the state of a device. (C/MM) 1284.4-2000

device coordinate system (computer graphics) A device-dependent coordinate system in which the coordinates of addressable points are expressed in integer addressable units. *Note:* A device driver maps normalized device coordinates or world coordinates to actual device coordinates. (C) 610.6-1991w

device-dependent (computer graphics) Pertaining to that which can be used only on a particular device. *Contrast:* device-independent. (C) 610.6-1991w

device driver (1) (computer graphics) The software that translates device-independent commands into device-specific commands. (C) 610.6-1991w
(2) The software responsible for managing low-level I/O operations for a particular hardware device or set of devices. Contains all the device-specific code necessary to communicate with a device and provides a standard interface to the rest of the system. *See also:* firmware device driver; operating system device driver. (C/BA) 1275-1994
(3) A program that runs on the host and manages the sending and receiving of information from the peripheral. The driver utilizes the link level interface defined in this standard to communicate data between the application program and the peripheral personality. (C/MM) 1284-1994
(4) A software component that permits a system to control and communicate with a peripheral device. *See also:* printer driver; disk driver. (C) 610.10-1994w

Device ID A structured, variable length ASCII message identifying the manufacturer, command set, and model of the peripheral. The message is provided by the peripheral in response to a request issued by the host during the negotiation phase. Provided that the peripheral supports the bidirectional mode requested by the host, this message is provided in the requested mode. The Device ID is intended to assist the host in selecting the device and/or peripheral driver appropriate to the peripheral. (C/MM) 1284-1994

device-independent (computer graphics) Pertaining to that which can be used on a variety of devices. *Contrast:* device-dependent. (C) 610.6-1991w

device interface One of the interfaces specified in this standard that allows devices to be identified, characterized, and used to assist other Open Firmware functions such as booting. (C/BA) 1275-1994

device media control language (data management) A language that may be used to describe the physical la[illegible] organization of data within some physical storage m[illegible] (C) 610.5[illegible]

device node A particular entry in the device tree, [illegible] scribing a single device or bus, consisting of propert[illegible] ods, and private data. (A device node may have multi[illegible] nodes and has exactly one parent node. The root no[illegible] parent node.). (C/BA) 12[illegible]

device path A textual name identifying a device node [illegible] ing its position in the device tree. (C/BA) 1[illegible]

device register (A) An addressable register used to st[illegible] mation describing the device. *See also:* control [illegible]
(B) An addressable register used to store status and [illegible] information, and data for transmission to or from a [illegible] *Synonym:* device status word. (C) 610[illegible]

device rise time (photomultipliers for scintillation c[illegible] The mean time difference between the 10- and 90[illegible] amplitude points on the output waveform for full cath[illegible] lumination and delta-function excitation. DRT is me[illegible] with a repetitive delta-function light source and a s[illegible] oscilloscope. The trigger signal for the oscilloscope [illegible] derived from the device output pulse, so that light [illegible] such as the the scintillator light source may be emplo[illegible] (NPS) 39[illegible]

device space (computer graphics) The area defined by [illegible] dressable points of a display device. (C) 610.6[illegible]

device specifier Either a device path, a device alias, or a [illegible] path that begins with a device alias and ends with a [illegible] path. (C/BA) 1[illegible]

device status word *See:* device register.

device tree A hierarchical data structure representing the [illegible] ical configuration of the system. (The device tree de[illegible] the properties of the system's devices and the devices' [illegible] tionships to one another. Most Open Firmware elem[illegible] vices, buses, libraries of software procedures, etc.) are [illegible] and located by the device tree.). (C/BA) 12[illegible]

dew point The temperature at which the water vapor in [illegible] begins to condense, expressed in degrees Fahrenheit [illegible] Celsius (°C). (PE/IC) 11[illegible]

device port The physical connection points through whi[illegible] nals flow into or out of a device or where timing, sy[illegible] nization, and triggering control are accomplished. (SCC20) [illegible]

device type Identifies the set of properties and package [illegible] that a node is expected to implement. Specified b[illegible] "`device_type`" property. (C/BA) 127[illegible]

device under test (DUT) The device to be placed in a te[illegible] ture and tested. (C/TT) 145[illegible]

dew point temperature The temperature at which conden[illegible] of water vapor begins in a space. (IA/PSE) 241[illegible]

dew withstand voltage test A test to determine the abili[illegible] the insulating system to withstand specified overvoltage[illegible] a specified time without flashover or puncture while [illegible] pletely covered with dew. (SWG/PE) C37.100-1992, C37.[illegible]

dezincification Parting of zinc from an alloy (parting is the [illegible] ferred term). *Note:* Other terms in this category, such as [illegible] ickelification, dealuminification, demolybdenization, [illegible] era, should be replaced by the term parting. *See also:* pa[illegible] (IA) [illegible]

DF *See:* direction finder.

DF antenna *See:* direction finder antenna system.

DFD *See:* data flow diagram.

D Filter A 300 Hz to 3400 Hz bandpass filter used for mea[illegible] noise, impulse noise, or data modem signal power. [illegible] measured through the D-Notched filter is used to evaluat[illegible] effect on the performance of a data modem. (COM/TA) 74[illegible]

D flip-flop A flip-flop that has one data input, one trigger, [illegible] an output which assumes the state of the data input when [illegible] trigger is received. (C) 610.10[illegible]